RADC-TR-88-155
Final Technical Report
July 1988

DISTRIBUTED C^2 SYSTEM RECOVERY MECHANISMS

Honeywell Inc.

Anand R. Tripathi, Helmut K. Berg, Jonathan Silverman, William T. Wood, Elaine N. Frankowski, Pong-Sheng Wang, Shiva Azadegan, Shiv Seth and Rita Wu

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-155 has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-88-155

6a. NAME OF PERFORMING ORGANIZATION
Honeywell Inc.

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COTD)

6c. ADDRESS *(City, State, and ZIP Code)*
Computer Sciences Center
1071 Lyndale Ave (South)
Bloomington MN 55420

7b. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Rome Air Development Center

8b. OFFICE SYMBOL *(If applicable)*
COTD

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-82-C-0154

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
63728F	2530	01	17

11. TITLE *(Include Security Classification)*
DISTRIBUTED C^2 SYSTEM RECOVERY MECHANISMS

12. PERSONAL AUTHOR(S) Anand R. Tripathi, Helmut K. Berg, Jonathan Silverman, William T. Wood, Elaine N. Frankowski, Pong-Sheng Wang, Shiva Azadegan, Shiv Seth, Rita Wu *

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final	FROM Sep 82 TO Aug 84	July 1988	178

16. SUPPLEMENTARY NOTATION *Subcontractors: Information Research Associates - Authors are James C. Browne, James Dutton, Vincent Fernandes, Annette Palmer, and Raj Kumar Velpuri
University of Texas at Austin - Authors are Donald I. Good, Michael K. Smith, (over)

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	07	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Distributed Systems, Reliable Systems,
Performance Evaluation, Reliability Evaluation,
Recovery Mechanisms, Atomic Action . [illegible] (over)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
This report describes an effort to develop a system designers guidebook for designing reliable distributed command and control systems. The guidebook contains a synthesis of reliable system design principles and methods to evaluate distributed system designs for performance, reliability, and functional correctness. The approach to developing the system designers guidebook in this effort is example driver. We develop a detailed design of a reliable distributed operating system and evaluate its performance.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas F. Lawrence

22b. TELEPHONE *(Include Area Code)*
(315) 330-2158

22c. OFFICE SYMBOL
RADC (COTD)

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 16 (Continued)

Fault-Tolerant Systems
Verification
Commit Protocol
Object-Oriented Systems
Volidation
Replication
Design Methods
Formal Specification

Block 16 (Continued)

Richard M. Cohen, Lawrence Smith, Lawrence Akers, William Bevier, Miren Carranza, Ann Siebert

CONTENTS

Page

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CHAPTER 1

INTRODUCTION

Reliability and timeliness are the two most important and critical attributes of command and control systems. The command and control system functions involve control of defense system resources and communication of intelligence and command information among the various constituents of the system. This requires timely collecting, processing, and communicating large amount of information to ensure effective coordination among the geographically dispersed components of the command and control system.

Distributed system technology provides an important and attractive approach to supporting the operations of the future command and control systems. This technology potentially supports:

1. Dispersion of the data as well as the processing functions to various locations of the command and control system.

2. Redundancy of data and functions to improve the reliability of the system due to the multiplicity of processing resources.

3. Efficient communication of information by networking of processing resources.

The goal of this contract is to synthesize a set of techniques for building reliable distributed systems for command and control applications and to evaluate their designs for fault-tolerance, reliability and performance. This work involved study of the system recovery mechanisms for distributed systems, development of concepts for integrating them into a distributed operating system, and finally a set of methods for evaluating the performance and reliability of such designs. The final outcome of this contract is a two-volume system designers guidebook titled A DESIGNERS GUIDE TO RELIABLE DISTRIBUTED SYSTEMS. The first volume of this guidebook presents the design and analysis methods, and the second volume contains the detailed designs of an example distributed operating system called Zeus and and its performance evaluation data.

The design of a distributed system involves many complex decisions. The purpose of a designers guidebook is to help a designer in systematically addressing the various design issues and making the most appropriate decisions so that the final design meets the desired requirements. It is important to stress the distinction between a guidebook and a handbook. A guidebook provides a comprehensive set of procedures which can aid a designer in achieving a goal. A handbook provides a comprehensive set of results (e.g., tables) which provide a basis from which a designer may make design decisions for a specific application. It is appropriate to write a handbook if one has the details of a set of applications of interest and the associated system environments. A guidebook is applicable to a larger set of problems and designers because of its orientation to procedures rather than results.

A guidebook describes the steps which take a designer from a set of requirement statements to a detailed system design which would exhibit the desired operational characteristics in a specified implementation base. Each design step refines the design and further defines what are the system's operational attributes. One set of attributes are those associated with the fault tolerance of a system -- availability, reliability, and survivability. An example of the design decisions that must be made are the degree of availability required for a given application and the performance required of a system environment to achieve it. It is a well-established principle that the designs should be subjected to early evaluations before starting any implementations. In fact, the design steps and the evaluation steps should proceed in a closely coupled fashion. This book presents a set of design guidelines for constructing fault tolerant distributed systems and a set of procedures for evaluating the desired operational characteristics of such designs.

The main contribution of this research is a unified presentation of system recovery mechanisms, a framework for their integration, and a set of evaluation techniques. It provides a starting point for the development of a design methodology of fault-tolerant distributed systems.

The system designers guidebook is organized into two volumes. The first volume describes reliability mechanisms, a framework for expressing designs, and techniques for evaluating mechanisms. There are two classes of problems that are not addressed in the reliability mechanism discussion -- security and Byzantine agreement. These problems were deemed outside of the scope of this contract. A framework based on object-oriented design is defined and used for expressing designs because it motivates the discussion of reliability mechanisms and aids in their integration into a unified design model. An example distributed operating system called Zeus is derived from the framework and used as a basis for presenting and demonstrating analysis techniques. This example design illustrates the integration of recovery mechanisms into distributed system designs. Zeus should be regarded as a design framework rather than a point solution. Although the mechanisms, techniques, and results are described within the context of an object-oriented design, they are equally applicable to process-oriented designs. Volume II contains the complete details of the example system and the results of its analysis. Some familiarity with Ada [DoD83](1) and the Concurrent System Definition Language [FRAN83a] is required to understand the detailed designs. The definition of the Concurrent System Definition Language is included as an appendix to this report.

(1) Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

INTRODUCTION

The approach taken in developing the system designers guidebook is depicted in Figure 1-1. The system recovery mechanisms are integrated into the Zeus design. Concurrent System Definition Language (CSDL) is used for the formal definition of the designs. PAWS (Performance Analysts Workbench System), Gypsy, NetRAT, and Path Pascal are used to evaluate the Zeus designs. The methodology used in the design, the analysis process, and the evaluation results are documented in the system designers guidebook.

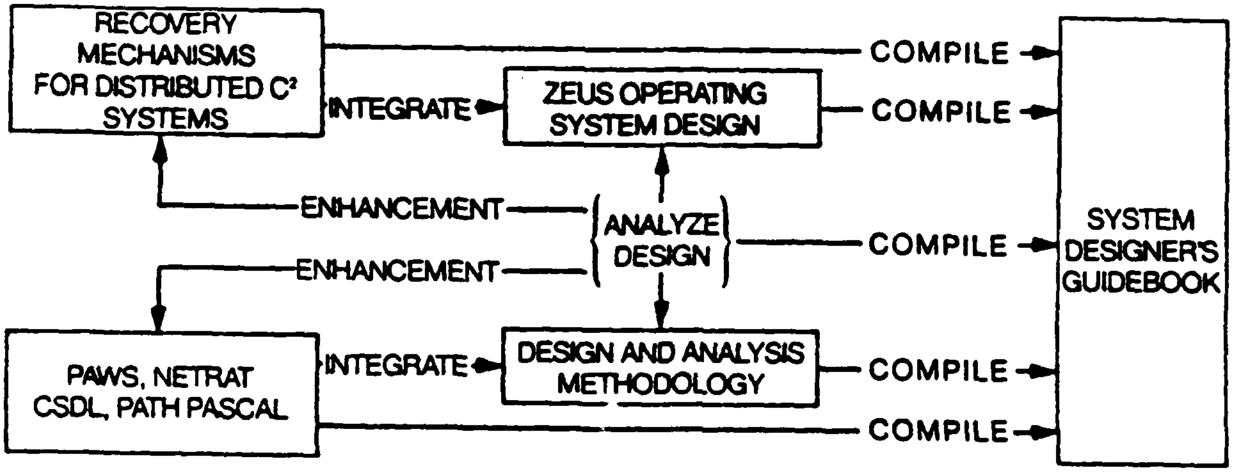

RECOVERY MECHANISMS FOR DISTRIBUTED C^2 SYSTEMS
INTEGRATE
ZEUS OPERATING SYSTEM DESIGN
COMPILE
COMPILE
ENHANCEMENT
ENHANCEMENT
ANALYZE DESIGN
COMPILE
SYSTEM DESIGNER'S GUIDEBOOK
PAWS, NETRAT CSDL, PATH PASCAL
INTEGRATE
DESIGN AND ANALYSIS METHODOLOGY
COMPILE
COMPILE

Figure 1-1. Approach to Distributed C^2 System Recovery Mechanisms

This report presents a concise yet complete overview of the technical approach taken during the course of this program and the highlights of the important technical accomplishments. Each chapter of this report describes an important milestone in the course of this contract, our approach in achieving the milestone, the uniqueness of the approach, and finally the major accomplishments. One of the highlights of our approach in developing the system designers guidebook is the definition and design of an example distributed operating system and the application of a set of design evaluation techniques using this example system as a testbed.

A system design can only be done in the context of its application environment. For this reason we pursued the task of studying the operational environment and the functional requirements of distributed command and control systems. These results of our study are described in the second chapter of this report. The main emphasis of this study was on the recovery mechanisms for distributed command and control system. One of the tasks was to survey the these recovery mechanisms and provide a comprehensive description of the mechanisms in the system designers guidebook. An important outcome of this task was an object-oriented design model for building reliable distributed systems. This design model integrates the surveyed recovery mechanisms into one framework. A brief overview of the survey is presented in Chapter 3. The object oriented design model and the example system, called Zeus, which is based on this model, are described in Chapter 4 and 5. Chapter 5 is devoted to the detailed designs of the example system and the formal definition of such designs. In this context we discuss the work performed on Concurrent System Definition Language (CSDL). The description of various analysis tools/techniques form an important part of the system designers guidebook. In this contract work we focused on using PAWS (Performance Analysts Workbench System)(1) for performance evaluations, NetRAT for reliability evaluations, Gypsy for formal correctness proofs, and Path Pascal for functional simulation for validating fault-tolerance. Chapter 6 describes the highlights of our work in the development and application of these techniques to reliable distributed systems. Chapter 7 presents the goals of the performance modeling of the Zeus system using PAWS, approach for evaluations, and the summary of the simulation results. Some of the possible future directions for this work include a detailed study of recovery mechanisms in a process-oriented design, design of fault-tolerant real-time systems, experimental evaluations of recovery mechanisms in the context of the object oriented framework developed under this contract, or development of an object oriented general purpose distributed operating system such as Zeus. Chapter 8 is devoted to the possible future directions for this work.

(1) PAWS is a registered trademark of Information Research Associates, Austin, Texas.

CHAPTER 2

DISTRIBUTED COMMAND AND CONTROL SYSTEMS

A system design can be meaningfully and successfully carried out only in the context of its intended application environment. A thorough understanding of the applications is essential in order to make the requirement statements for the system functionality, reliability, and performance. The system functionality and the associated reliability statements identify the kinds of failures the system must withstand and the consistency that must be maintained for the system objects in the presence of failures and concurrent operations. The performance statements identify the desired response time and throughput of the various system functions under the specified workload. The results discussed here describe the operational characteristics and the functional requirements of distributed command and control systems, and also identify the forms of requirement statements for system performance and reliability. The results of this effort are applicable to both strategic and tactical command and control systems.

2.1 Command and Control Systems

Any command and control system must support four basic functions: communication, navigation, data collection and decision support. These systems can be divided into two broad categories, strategic and tactical. Systems in these two categories differ in the geographic scope of the system, their functional complexity and the mobility of the system nodes. Strategic command and control system encompass a relatively large region of operations (roughly 500 to 1,000 miles radius). It maintains large long-lived databases and contains several smaller, and possibly tactical, command and control systems as its constituents. Tactical command and control systems are generally smaller in geographic scope; the distance between nodes is typically 10-200 miles. The nodes of the tactical systems are relatively mobile - they can be moved and installed in a few days. The communications facilities that connect nodes of a tactical system are usually much less reliable than those used to connect the nodes of strategic systems.

2.1.1 Command and Control System Function

The general goals for the data processing elements in a command and control system are to:

- o Make information available to the users who need it.
- o Improve the response time of time-sensitive operations.

- o Support the database needs of the users.
- o Make available global databases which are needed for planning, coordination, threat assessment, targeting, intelligence production and status monitoring.
- o Provide reliable dissemination of messages carrying requirements, commands, warnings and status information.
- o Provide extensive degraded mode operating capability.
- o Provide enhanced survivability and continuous operation under the loss of C2 system components.
- o Support multi-user multi-level security of information.

Efficient database sharing is the most crucial requirement of command and control systems. A command and control system must support global logical objects for the following kinds of information: weather, personnel, logistics, enemy situation, friendly situation, surveillance and identification, warnings and alerts, mission status, tactical air support requests, etc. The major role of the data processing functions performed by a command and control system are concerned with maintaining this data base and providing timely and accurate reports using the database.

2.1.2 Operational Environment

Because of the evolutionary nature of future distributed C2 systems, it is desirable to adopt an approach which permits relatively easy changes for system expansions, capacity upgrades, functionality upgrades, hardware substitution, and addition of new elements. The approach of modular system design should also help in rapidly configuring new systems.

Instead of designing systems to meet certain specific requirements, it is desirable to provide an architecture which can adapt easily to the long term changing requirements due to the state of the technology and the world-situation, as well as the short-term changes in requirements due to the tactical environment. One can use physical and communication environment features of distributed command and control systems to characterize their operational environment.

Physical Environment:

A single command and control system consists of several geographically dispersed command centers. The distance between the units can range from a few miles to a few hundred miles for tactical systems, and up to a few thousand miles for strategic systems. The geographical dispersion serves to increase the field of view or to provide higher survivability to the command centers by locating them in rear areas.

Communication between the command centers in a C2 system can be implemented with microwave or radio frequency channels. In some cases, where the distances are not too large, coaxial cables or fiber-optics cables may be used. The command centers are high value targets, and placing them in the rear areas for reasons of survivability will decrease the performance because of communication delays.

Most of the communication among and within constituent command centers of a command and control system consists of command messages, and database updates and query messages. Most of the other data processing requirements of a command center will normally be supported by the resources co-located within it.

Most of the important databases critical to the command and control operations are maintained at the command centers. To support continued operations in the event of the loss of a command center, another center must be able to reconstruct the database from the replicated components of the global database.

Communications Environment:

The communications within a command center will be, in general, supported by a local area network (LAN) and that among the command centers will be supported by long-haul networks. Thus a long-haul network connects several LANs in a C2 system. The long-haul network topology will change dynamically because mobile command centers will be moved in response to the tactical situation. The length of the a LAN communication link can range from a few meters to hundreds of meters. The long-haul links can be up to about thousand miles long. The communication bandwidth for LANs ranges between 1-10 mb/sec, and for long-haul communication from 10-50 kb/sec.

The long-haul network must be connected to external elements such as the World Wide Military Command and Control System (WWMCCS), Intelligence Data Handling System Communications (IDMSC) and the Defense Communications System (DCS).

Some of the biggest problems which will affect the communications system performance are electronic warfare, self-jamming, and the loss of nodes (mininets). Network partitioning, node drop-out, node reunion, network reconfiguration are some of the problems which the designers must address.

2.2 Architecture Of Distributed Systems

Redundancy of both hardware and software resources is the most important characteristic of reliable systems that support continued operations despite component losses. The geographical distribution of critical system databases and processing resources is key to the design of survivable systems. Thus, in the event of loss of a particular site in a command and control system, it should be possible to use database copies and processing resources at other sites. The need to maintain and update replicated databases imposes the following requirements for the underlying architecture:

The system must contain appropriate processing resources (CPU, memory, secondary storage) at each site.

There must be communication between the sites as well as with local and remote users of the databases (a) to keep the replicated copies mutually consistent and (b) to provide access to remote users on system reconfiguration.

These two requirements make distributed system architectures the most natural candidates for supporting highly survivable C2 systems. Functional redundancy and geographical dispersion enables distributed systems to survive hostile actions and to provide continuous operation. These advantages of distributed systems arise partially from the distribution of system state information. However, effective survivability mechanisms are based upon consistent system state. Distributed operating systems used in this application must incorporate mechanisms to maintain the consistency of the distributed system state information in the presence of concurrent updates and system component failures. This is essential to guarantee correct functioning on reconfiguration and restart; therefore, suitable recovery mechanisms and concurrency control mechanisms are required in the distributed operating system to maintain consistency of distributed state variables.

A distributed system consists of multiple computers interconnected by a communication network that cooperate to complete a computation. The mechanisms that enable the cooperation are implemented by a distributed operating system. A conceptual picture of a distributed operating system is shown in Figure 2-1.

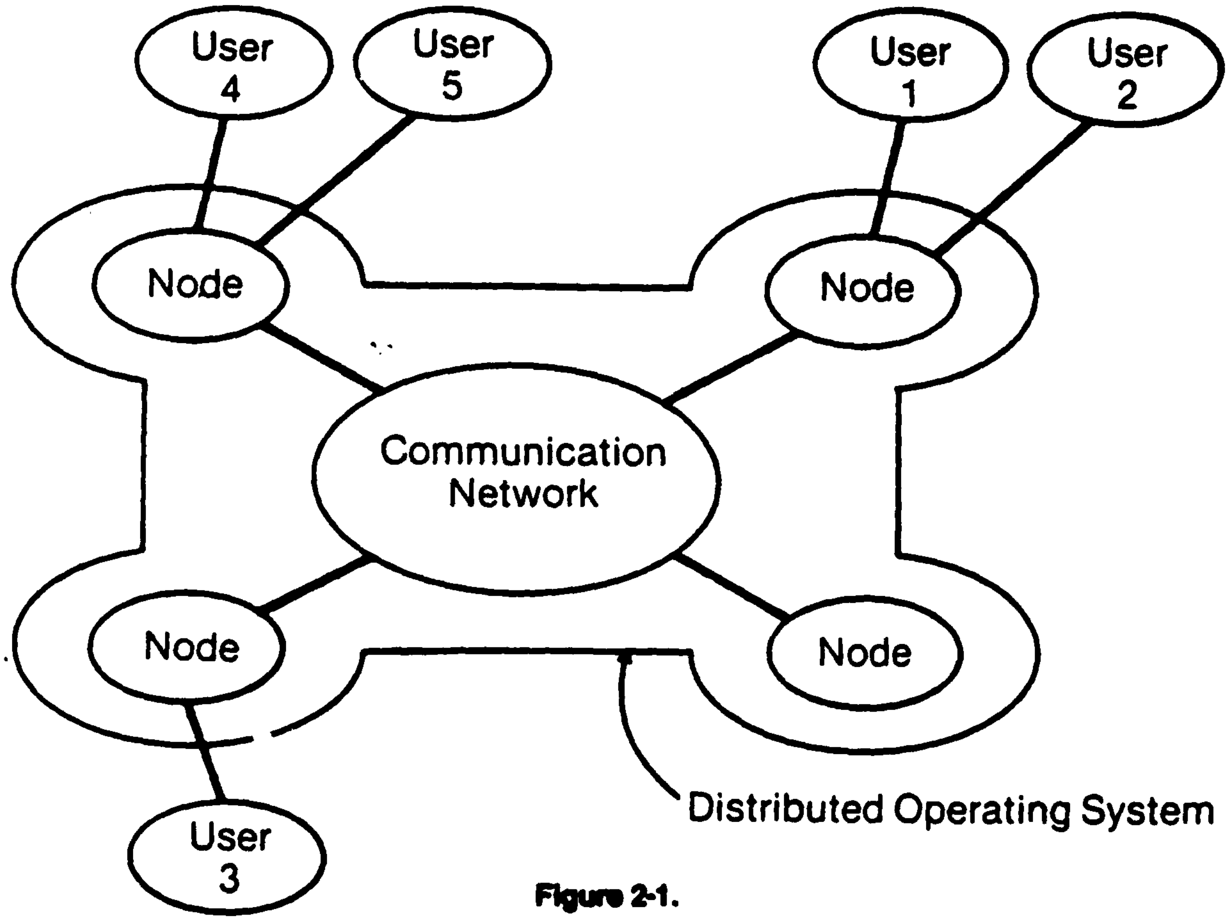


Figure 2-1.

A site consists of at least one physical processor, an operating system kernel, primary and possibly secondary memory, an interface to the communication network, and possibly interfaces to input/output facilities. The sites are physically separated and communication occurs by message exchange rather than by shared memory. Each site has processes and resources which constitute fragments of system processing activities. Since control of these processing activities is distributed among the sites, a single site normally has neither system-wide authority nor a complete view of the global system state.

A distributed operating system creates and manages logical (perhaps physically distributed) resources (processes and files) and physical resources (processors and memories). A distributed operating system is based on a set of protocols which govern interaction between sites. The operating system kernel at each site manages its physical resources autonomously and may cooperate with other kernels in the management of its logical resources. The state information may be partitioned and distributed among the operating system kernels. The individual kernels operate concurrently, and possibly asynchronously on the basis of locally available state information. The system interface consists of a set of functions which the distributed operating system provides to the application environment. Ideally, these interfaces should have the following features:

1. Transparency of resource locations: The user-visible functions for accessing the resources in the system should make transparent to the clients the location of the resources. The mechanisms for accessing the remote and the local resources should be uniform.

2. Transparency of recovery mechanisms: The interfaces provide by the distributed operating system should make the recovery mechanisms transparent to the clients; however, the clients should have enough control over the selection of those parameters that are critical to the system performance in different application environments.

A communication system transfers information among the sites in a distributed system. It is used by distributed operating system kernels, system processes, and application processes to convey updates and to gain access to global system state and to utilize resources provided by other sites. Communication systems typically appear in system designs and implementations at a higher level of abstraction because the detailed realization is isolated from the remainder of the system. In the context of a command and control system, a communication system is meant to include an internet, a collection of interconnected networks.

2.3 Reliable System Requirements

An application may be described as a collection of objects on which operations are executed by users. Some operations may be combined together and executed as a single end-user operation. A requirements statement may be made about the performance and reliability of the operations as described below.

Performance Requirements: In general the performance requirements are specified in terms of the response time and throughput. There are several ways in which these two measures may be specified for a distributed system. Average throughput and average response time for the execution of a given operation on an object can be specified in the following terms:

(1) For the overall system,
(2) For some particular sites in the system,
(3) For each operational mode such as emergency/peace-time modes,
(4) As a function of available resources (sites) in the system.

In addition to the mean values, upper and lower bounds or variances may be specified for these measures.

Reliability Requirements: Traditionally the reliability requirements for a service or operation are specified in terms of its expected availability and mean-time-to-failure. Like the performance requirements specifications, the reliability requirements can also be specified in terms of the four ways described above. It may also include the types of failures that must be withstood, and the number of failures of a given type that must be withstood.

Similar requirements may be made about groups of operations. In addition, it is assumed that statements are made about what the hardware configuration is and the assignment of objects and operations to sites. From such information we are interested in answering two questions: "What level of reliability does a system provide?", and "What is the extra cost of the reliability?".

Ideally a user should be able to specify the desired reliability requirements for objects, operations, and groups of operations without knowing the details of the implementing mechanisms; tools should then automatically configure a system. More realistically, a system administrator who is knowledgeable about the system's hardware and software will manually make the selections and adjustments needed to achieve the desired level of reliability.

In order to state system requirements and to develop a system that meets them we are still faced with a problem of how to specify the requirements. Traditionally, component reliability is given by statistical quantities such as the mean time to failure (MTTF), mean time to repair (MTTR), and the probability of availability. This suggests that one way for a user to specify the reliability of objects and operations is to give the desired values (e.g.,0.95 MTTF). This is certainly the most accurate way of defining the expected reliability of an object since it takes into account the interrelation among all of the dependent objects and components of a system and their individual characteristics. There are, however, at least three problems with using statistical quantities for user level specifications.

First is the problem of using small quantities to specify values. Should the MTTF be 0.94 or 0.95? Why? Would different users choose different values for similar objects?

Secondly, there are typically many different combinations of parameters, replication strategies, and configurations which will yield the same or roughly the same MTTF and availability for a given object. A simple numerical quantity gives no indication as to which of several possible strategies to choose. Furthermore, without being given additional information, it is difficult for an administrator and probably impossible for the system itself to choose the appropriate solution.

The third problem arises due to the application environment which is being considered in this guidebook. The probabilities of component failures may change unpredictably under military stress conditions. The MTTF and availability of a component completely describe its fault characteristics. There are many circumstances, however, where these metrics are difficult or impossible to evaluate. As an example of such a circumstance, consider a command and control system in a potential combat situation. The definition of a component "fault" in such a system would have to include the destruction or disruption of that component in combat; and thus, this eventuality must be taken into account when calculating the MTTF and availability of the component. Unfortunately, the probability of an attack, or the probability that an attack will ensue in such a way as to affect the performance of a given component of the system, depends on a number of decidedly non-quantifiable factors such as political climate, human factors, recent history, and so on. In such conditions it is more reasonable to ask questions such as "Does this service (function) remain available given that a set of components are unavailable?".

The alternative to using statistical quantities provides a set of pre-defined reliability levels. Associated with each reliability level is a consistency (integrity) specification. The levels overcome the problems with strictly numerical specifications by associating a boolean-valued consistency requirement. An object is said to belong to a certain reliability level for a given set of faults if the associated consistency specifications are maintained under the presence of those faults. The object is viewed as being "completely immune" to the faults in that set for the associated consistency level. The maximum cardinality of such a set for a given reliability level of an object determines the robustness of that object. Faults may include events such as site failures, link failures, disk failures, and memory failures. Each category can be refined when it is appropriate to do so. For example disk failures can be refined to include single page failures and disk pack failures.

2.4 Design Issues And Tradeoffs

Effectiveness of a recovery mechanism can be measured in terms of recovery time and performance overhead. To see why the recovery time and performance overhead are important in evaluating the recovery mechanism, consider the performance of a system under normal and faulty conditions. Assume that throughput (defined as the number of units of work performed per unit of time) is an indication of the system performance. If there were no failures, there would be no need for recovery mechanisms. In this hypothetical situation, under a constant load (e.g., fixed number of jobs running in the system at all times) the throughput stays constant at a level that is referred to as the ideal level of performance. Introducing recovery mechanisms into this system to enable it to deal with failures degrades the performance even when there are no failures. An operating overhead is imposed equal to (1) the processing overhead required to check and maintain information about system state for recovery and (2) a storage overhead equal to the storage required to hold redundant information. It is desirable to choose those recovery mechanisms that have the least performance overhead under normal operation.

When an error condition occurs, certain recovery procedures are initiated. These procedures cause an even higher performance overhead. This is called failure recovery operation overhead. After the fault is eventually cleared and the system is recovered, the performance goes back to the level before the failure. Figure 2-2 depicts this simplified situation. There are two important parameters that have to be considered when a failure occurs. First, how much time does it take for the system to recover from the failure? This period of time is called system recovery time. For the duration of the system recovery time, the performance of the system is at its lowest level. Therefore, a good recovery mechanism has to minimize this time period. Second, how much is the performance of the system degraded for the duration of the system recovery? The performance overhead factor includes both the normal operation overhead and the failure recovery overhead.

A more realistic situation is depicted in Figure 2-3. The system operates normally until a fault occurs and some component of the system becomes inoperative. The system executes recovery procedures and operates at reduced capacity. After the recovery procedures are executed, the performance rises to a level below full system performance. Later, the fault is cleared and the system executes recovery procedures to restore the consistency of global system state. During this time, performance is again degraded. Finally, throughput is restored to the normal level.

This view introduces additional effective measures: Reduced configuration overhead is the difference between ideal performance and performance while part of the system is inoperative. Reconfiguration recovery overhead is the difference between ideal performance and performance while global system state is being restored. Reconfiguration time is the duration of this processing.

Cost is another important factor in deciding which recovery mechanisms should be included in a distributed system. Cost may be measured in terms of the additional hardware resources required to implement a recovery mechanism while maintaining the same level of performance as without the recovery mechanisms. This includes the cost of additional primary and secondary storage and processing power. The memory requirement is derived from the size of the recovery mechanism procedures and the size of any additional data structures. The secondary storage requirements may be further increased if they are required to store multiple copies of objects. The additional processing overhead is derived from the performance overhead previously discussed. Another way to characterize the overhead due to recovery mechanisms is in terms of reduction in response time and throughput.

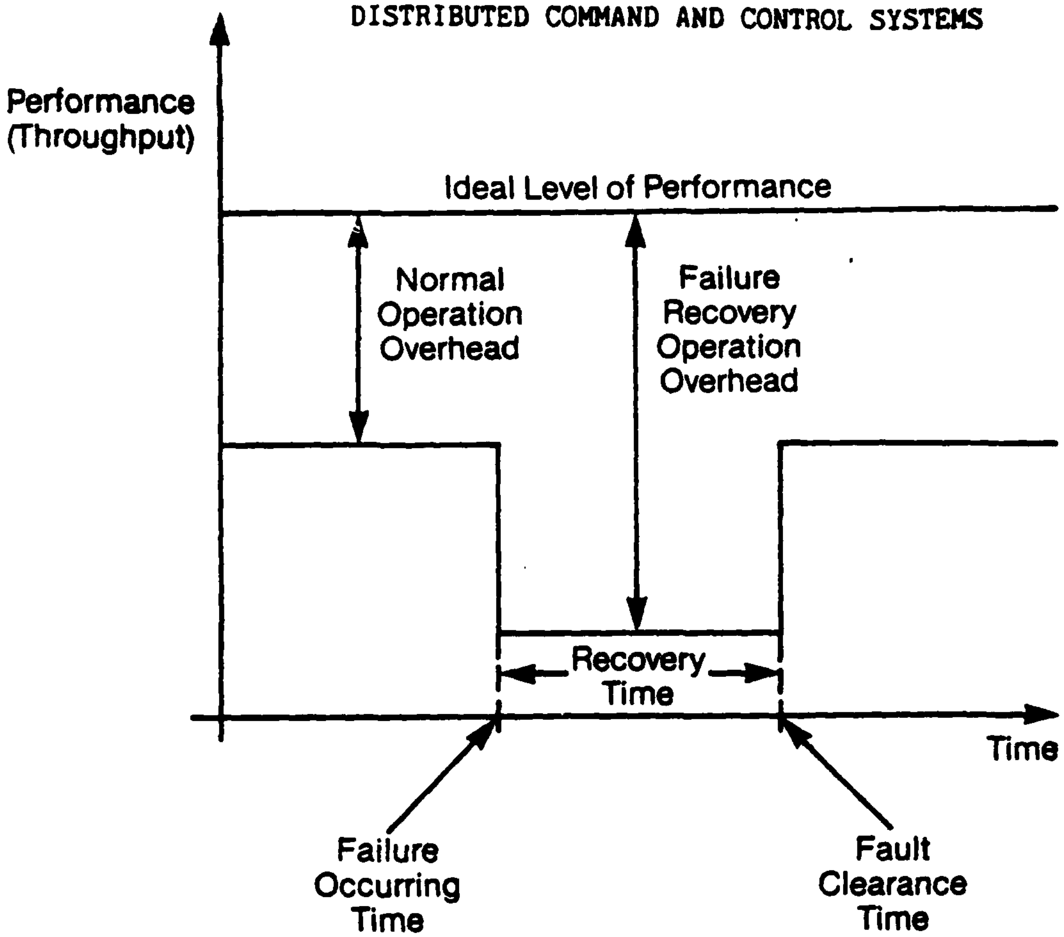

Performance (Throughput)
Ideal Level of Performance
Normal Operation Overhead
Failure Recovery Operation Overhead
Recovery Time
Time
Failure Occurring Time
Fault Clearance Time

Figure 2-2.

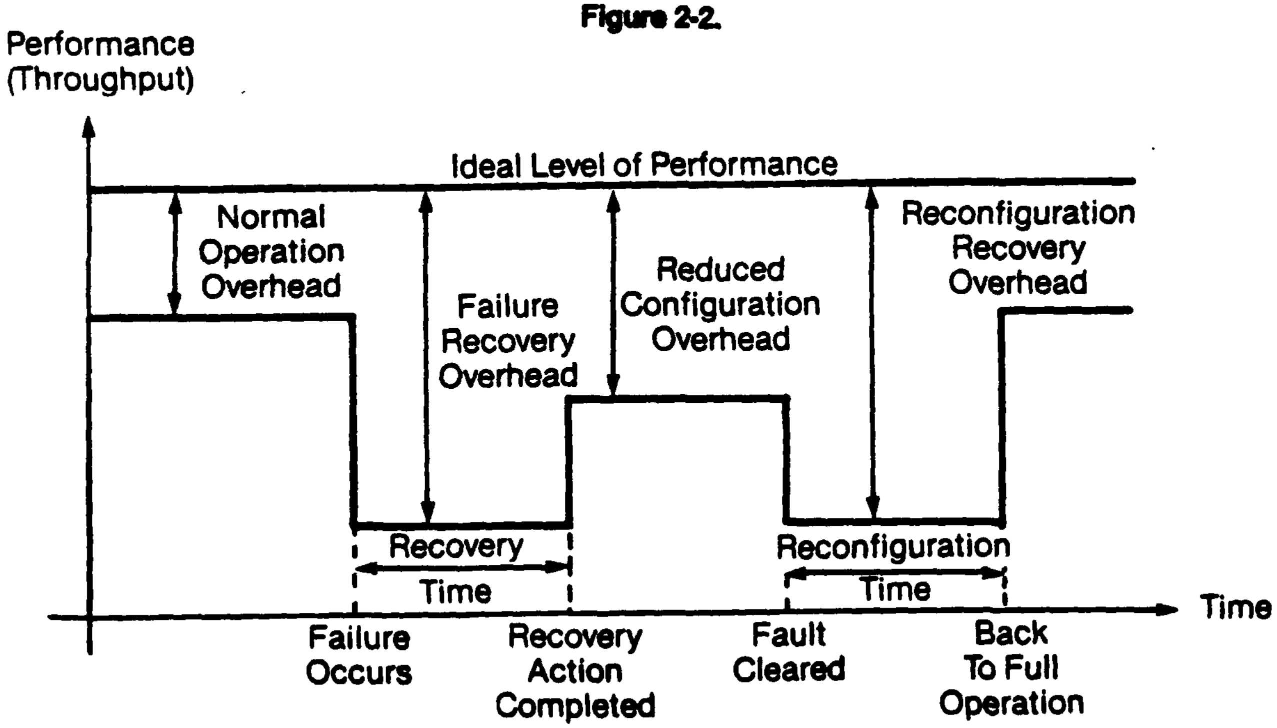

Performance (Throughput)
Ideal Level of Performance
Normal Operation Overhead
Failure Recovery Overhead
Reduced Configuration Overhead
Reconfiguration Recovery Overhead
Recovery Time
Reconfiguration Time
Time
Failure Occurs
Recovery Action Completed
Fault Cleared
Back To Full Operation

Figure 2-3.

2.5 Summary

The system designers guidebook presents the results of our study of functional requirements of distributed command and control systems. In designing such systems it is important to understand their operational environment in order to define the performance and reliability requirements. Distributed system architectures seem to be the most ideal and natural choice for implementing the future command and control systems. This is due to the fact that such architectures support integration of geographically dispersed processing elements into one coherent monolithic system. This integration is achieved by a distributed operating system which provides mechanisms for managing distributed resources in the system. Transparency of resources and the recovery mechanisms for the resource management functions are two important attributes of an ideal distributed operating system. Introduction of recovery mechanisms introduces certain performance penalties such as reduced throughput and response time because of extra resources and CPU cycles required for maintaining additional system state needed for recovery.

CHAPTER 3

INTEGRITY MECHANISMS

The operating conditions that exist in a distributed system define requirements for the consistency and reliability management techniques. These conditions include concurrent operations and component failures. Concurrent operations may access common data and inadvertently compromise the integrity of the data. If there are multiple copies of data, the problem of concurrent access must address the issue of the interdependency of the copies values. Whenever components fail or if users are permitted to abort operations all other sites that are executing a part of the operation must be informed and data restored to a consistent state. If a user's computation is dependent on an intermediate value of a failed or aborted computation, the dependent computation may also have to be rolled back. It is possible that a cascade of rollbacks, a domino effect, may occur.

The consistency requirements in a distributed system are characterized by four criteria. The first criterion, internal consistency, is the semantic integrity of the data. The second criterion, mutual consistency, is the relation between the copies of replicated distributed data. One example of a mutual consistency requirement is that all copies of a replicated data converge to the same value sometime after the updating of data is stopped. The third criterion, external consistency, is the relation of the system interactions with the users. For example, if a user invoking a transaction is given a response indicating successful completion, then the updates made by the transaction must be reflected in the database. The external consistency requirements are dependent upon the definition of the user-system interface. Interactive consistency [LAMP82], the fourth criterion, requires that all correctly functioning nodes in the system have an identical view of the system despite the malfunctioning of some nodes. This is also known as the Byzantine Generals Problem.

The key principles for designing reliable systems are the atomicity of transactions and the management of redundant components. A transaction is a set of primitive operations on data that appears to be executed as an indivisible operation. A transactions implementation in a distributed system requires a protocol by which a collection of processes may reliably decide to make permanent (i.e. commit") its effects. Transactions provide a common work unit for the problems of error recovery and synchronization. The techniques to solve these two problems in a design interact closely with each other because of the need to maintain recoverable consistent states.

Management of redundancy in the system in the form of replication of objects or creation of backup objects is important for supporting continued operations in the event of loss of resources. The major problem in redundancy management is the maintenance of consistency among replicated objects, and the maintenance of sufficient up-to-date state information with the backup modules to support reconfiguration. Several strategies may be used to manage such state information, for example keeping a majority or a survivable set of the replicated units in a consistent state.

This chapter introduces the terminology, concepts, and issues involved in consistency and reliability management techniques. The details of the algorithms for implementing the techniques are contained within the system designers guidebook.

3.1 Consistency Management In Distributed Systems

The goal of concurrency control techniques is to maintain mutual, internal, and external consistency requirements of shared data and to maximize the throughput of access to the data. The techniques used for maintaining consistency of data under concurrent update operations consist of four tasks. The first is to assign an order to all the transactions. The second is to identify conflicting transactions and conflicts. The third is to realize the inter-site synchronization required to achieve this order for the conflicting transactions. The fourth is to achieve the required intra-site synchronization. The schedule produced may be serializable or non-serializable. A serializable schedule means that the final effect of executing interleaved operations of concurrent transactions on tne database is equivalent to some serial execution order of those transactions. A non-serializable scheduler seeks to increase the concurrency between transactions by examining the semantics of operations. A serializable scheduler uses only the syntax of a transaction. Almost all systems to date use serializab:e schedulers. There are three basic techniques to achieve serial consistency: timestamps, locks, and optimistic.

3.1.1 Timestamp Based Protocols

In timestamp-based protocols, every transaction and every data item is assigned a globally unique timestamp. The timestamp of the datum is equal to the timestamp of the last transaction that accessed the datum. To access a datum, a transaction sends its timestamp and the type of operation (e.g., read or write) to the site where the datum resides. In order to serialize requests and resolve conflicts a scheduler at the site where the datum resides uses a rule to compare the timestamp and operation request of the transaction with the timestamp of the datum.

A number of timestamp based protocols have been proposed. In general, the greater the amount of concurrency permitted, the greater the probability that an operation may be rejected and a transaction restarted. Basic timestamp-ordering and conservative timestamp-ordering are the endpoints of a spectrum. Basic timestamp-ordering delays operations very little, but it tends to reject many operations. It schedules a transaction's operation if its timestamp is greater than the timestamp of the datum. Conservative timestamp-ordering never rejects operations, but it tends to delay them often. It requires that a scheduler have an operation request from every other node before a request is granted. Since it has a request from all nodes, it can safely allow the request with the smallest timestamp to proceed.

3.1.2 Locking Protocols

In locking protocols, a transaction requests a lock on an object, operates on the object only when it has been granted a lock, and releases the lock on the object when it no longer needs the object. The exact time that a transaction releases a lock is dependent on how the logical database is organized. If the logical database has the structure of a directed graph a transaction may release an object as soon as its operation on the datum is completed, otherwise it must wait until there are no other locks to be acquired. The latter case is called two-phase locking and the former non two-phase locking.

Two-Phase Locking Protocols

A two-phase locking protocol specifies that in each transaction all the locking operations must precede any unlocking operation, and all transactions must be well-formed. A well-formed transaction acquires locks on objects before accessing them. It has been shown in [ESWA76] that if all transactions follow the two-phase locking protocol, then the schedules of their executions are serializable. In the two-phase locking it is easy to see that deadlocks are possible. To avoid deadlocks we could set an order to all the entities and stipulate that all the transactions request locks only in the set order. Alternatively, when a transaction has been permitted to start executing, it may put intention locks on all the entities it would ever need. These locks may be used to rule out the possibility of a deadlock before permitting any other transaction to set its intention locks.

Another approach to ensure deadlock freedom is to add a deadlock prevention scheme to the locking scheme. Rosenkrantz, et al., [ROSE78] have proposed two such deadlock prevention schemes. Timestamps are assigned to transactions and are used as priorities in determining what to do when a transaction requests a lock on an object that is already locked. In the Wait-Die scheme, an older transaction waits on the completion of a younger transaction that holds a resource that the older transaction requests; but a younger transaction that requests a resource held by an older transaction is forced to restart. In the Wound-Wait scheme, an older transaction waits on a younger transaction only if the younger transaction has started its termination; otherwise, the younger transaction is restarted. A younger transaction is allowed to wait on an older transaction.

The Wait-Die scheme has the disadvantage that a younger transaction may restart and die several times before completing successfully. The restarts will consume some of the system resources. However, this scheme has the advantage over the Wound-Wait scheme that after a transaction has acquired all of the resources it needs, it can not be pre-empted and restarted. In the Wound-Wait scheme, even when a transaction has locked all the resources it needs, but has not yet initiated its termination, it is possible that for it to be wounded and forced to restart.

Non-Two-Phase Locking

Only a few protocols have been proposed that are not two-phase locking. One of these, proposed in [SILB81] presumes a tree-structured, hierarchically organized database. Transactions must be well formed and locks are acquired as follows. A transaction Ti may initially request a lock at any node (e.g., entity). Subsequent lock requests may be made only for direct descendants of nodes for which Ti already has a lock. When a lock is released, it may not be reacquired. The schedules produced by this protocol are serializable and, unlike the two-phase locking protocols, are deadlock free. An intuitive understanding of this fact is straightforward. Each transaction has a frontier of lowest nodes in the tree on which it holds the locks. The protocol guarantees that these frontiers do not overlap. If the frontier of Ti begins above the frontier of Tj, it will remain so, and every item to be locked by both will be locked by Tj first.

When locks are used in a distributed system a number of additional considerations arise. Among them are is global synchronization required to lock an object, how is an object globally synchronized, how can global synchronization be achieved with a minimal number of messages, and how can any one node be kept from becoming a performance bottleneck and a single point of failure. These issues are discussed at length in the system designers guidebook.

3.1.3 Optimistic Concurrency Control

The optimistic method for concurrency control [KUNG81] hopes that transaction conflict is rare and that concurrency can be increased by eliminating locks and their associated overhead. Every transaction goes through three phases -- read, validation, and write. During the read phase, a transaction reads objects, creates local copies of the objects, and updates the local copies. The validation phase determines if the operations of a transaction conflict with those of another transaction and violate serial consistency requirements. If the test fails, the transaction is aborted. Otherwise a transaction enters a write phase where its updates are made permanent or the results of a query are displayed.

In order to ensure the serializability of the transaction, each transaction is assigned a unique integer and the transactions are serialized according to their assigned numbers. The validation procedure ensures that one of the following three conditions holds: 1) a transaction, Ti, with a smaller assigned number completes its write phase before a transaction, Tj, with a larger assigned number starts its read phase; 2)the write set of Ti does not intersect the read set of Tj, and Ti completes its write phase before

Tj starts its write phase; and 3)the write set of Ti does not intersect the read set or the write set of Tj and Ti completes its read phase before Tj completes its read phase.

Condition (1) states that Ti actually completes before Tj starts. Condition (2) states that the writes of Ti do not affect the read phase of Tj, and that Ti finishes writing before Tj starts writing, hence does not overwrite Tj (also, note that Tj cannot affect the read phase of Ti). Finally, condition (3) is similar to condition (2) but does not require that Ti finish writing before Tj starts writing; it simply requires that Ti not affect the read phase or the write phase of Tj (again note that Tj cannot affect the read phase of Ti, by the last part of the condition).

The transactions are assigned their transaction numbers after they complete the read phase to avoid the possibility of a more recent transaction with a short read phase being blocked by an earlier transaction with a long read phase. This scheme of assigning transaction numbers does not require the validation of condition (3) above.

3.1.4 Basic Timestamp Ordering Versus Locking

Timestamp ordering in centralized systems tends to behave very similar to locking but has the disadvantage of inducing larger numbers of restarts. This is because the timestamp ordering scheme a priori determines the serialization order. What may appear to be a transaction conflict that induces a restart based on timestamp ordering may not be a conflict using locking and optimistic methods. For example, if a transaction with a larger timestamp reads an object and completes before a transaction with a smaller timestamp writes the same object, the transaction with the smaller timestamp will be aborted. Locking and optimistic schemes would allow both transactions to complete successfully.

Locks are required to implement critical sections in both timestamp ordering and optimistic schemes. For example, locks are required while reading and updating the timestamps associated with the objects. More importantly, in the timestamp ordering scheme some form of logical locking is required to prevent triggered aborts. Such a situation arises when a more recent transaction is allowed to read objects that have been updated by a transaction that is uncommitted and later aborts. The more recent transaction is also aborted. To prevent such a situation, access to an updated object by other transactions is blocked until the updating transaction either commits or aborts. This is equivalent to holding a write lock on the object. It should be noted here that the optimistic scheme avoids locking and accepts the possibility of transaction aborts.

3.1.5 Non-Serial Consistency

It is only recently that researchers [GARC83b] [FISH82] [BLAU83] have started investigating consistency management techniques that exploit the semantic knowledge of the database during concurrency. Such a knowledge can lead to certain acceptable schedules that are not serializable. This area of research is relatively unexplored.

In [GARC83b] Garcia-Molina investigates how the semantic knowledge of an application can be used in a distributed database to process transactions efficiently and to avoid some of the delays associated with failures. In [GARC83], the main idea is to allow nonserializable schedules which preserve consistency and which are acceptable to the system users. To produce such schedules, the transaction processing mechanism receives semantic information from the users in the form of transaction semantic types, a division of transactions into steps, compatibility sets, and countersteps. Using these notions, in [GARC83], a mechanism is proposed which allows users to exploit their semantic knowledge in an organized fashion.

3.2 Reliability Techniques In Distributed Systems

In this section we review various error recovery techniques and their applicability in distributed systems. Our discussion of recovery techniques starts with a brief overview of the concepts and definitions in this area. Detailed discussions of these concepts and definitions can be found in some of the surveys, [RAND78] [KOHL81] [VERH78], in this area.

A system is said to have failed when it no longer meets its specifications. The transition into the failed state is characterized by the failure event. The term error is used to characterize an incorrect system such that any further computation activity using the normal algorithms would result in a failure of the system. A fault is the mechanical or algorithmic malfunction (i.e., failure) of a system component that may cause an erroneous state.

All reliability techniques are based on adding redundancy in the system to support recovery from errors and continued operation. This is called protective redundancy. It is manifested in a system as additional components, data, and algorithms. This section discusses the additional components, data, and algorithms necessary to do error detection and recovery in a distributed system.

3.2.1 Error Detection Techniques

The purpose of error detection techniques is to detect the erroneous states of the system that could lead to system failures. Some general techniques for error detection [ANDE79] are described below.

INTEGRITY MECHANISMS

(a) Replication Checks: In such schemes, an activity is replicated and the results from replicated activities are checked for consistency. An inconsistency among results indicates a possible error condition. Errors can be masked by majority voting as in Triple Modular Redundant systems.

(b) Reversal Checks: They involve application of inverse computation to check what the input to the system should have been. The calculated input and the actual input are compared for consistency.

(c) Coding Checks: They are the most popular error detection technique. Redundant information in the form of checksum or parity is associated with objects to detect erroneous states.

(d) Acceptance Tests/Consistency Checks: At certain well-defined points in the execution, tests are applied to the objects that define the state at that point. Such tests ensure that the state at that point conforms to certain specifications. Any inconsistencies imply an erroneous state. Consistency checks can also be applied to some mutilated data structures that are reconstructed on recovery.

(e) Interface Tests: These tests ensure that the interactions among system components meet certain acceptance criteria. Tests are applied to the parameters and the results of interface functions. Such tests limit propagation of errors from one component to another through the interfaces. The confinement of errors is strongly dependent on how rigorous the acceptance tests are. In distributed systems, interfaces provide well-defined and controlled means for the propagation of exception conditions between modules. If the interface function execution encounters error conditions, then an error condition is returned to the caller through the interface.

(f) Diagnostic Checks: In such techniques, explicit tests are conducted on system components for which expected outputs for given test inputs are known. The failures of components to be tested and the components conducting the tests should be independent. As pointed out in [ANDE79], diagnostic tests are rarely used as a primary error detection mechanism, rather used as a supplement to other detection mechanisms.

(g) Interval Timer/Time-Out Mechanisms: In distributed systems, time-out techniques are frequently used to detect possible error conditions. A process invoking a remote operation waits for a certain specified period (called the time-out period) to receive the response. If no response is received within this period, then an exception condition is raised and appropriate forward error recovery is initiated.

3.2.2 Error Recovery Techniques

Recovery techniques involve the generation of consistent system states. There are two categories of techniques: backward error recovery and forward error recovery. Backward error recovery techniques save prior consistent states in an execution history. When an error is detected recovery involves restoring a computation to a saved prior consistent state. Checkpoint/rollback is typical of these techniques. The forward error recovery techniques use the present computation and error state to arrive at a new consistent state. They are typified by programming language exception handlers. The techniques in the latter category are application dependent, while those in the former are application independent. We will restrict our discussion to backward error recovery techniques.

3.2.2.1 Checkpointing and Rollback

In this technique, the state of the process is saved on a stable storage as a checkpoint. A checkpoint is a backup version of the complete execution environment of the process. When the system is recovering from an error a checkpoint of the system (e.g., the state of all processes executing at a given time) is loaded and restarted.

The rollback of a process in a system of communicating processes may cause rollback of other processes. This happens when a process that is rolled back to a previous checkpoint has communicated some information to some other processes after establishing that checkpoint. Thus, all messages sent after that checkpoint are revoked, and all activities performed by the recipient processes after receiving such messages are invalid; this causes all recipient processes to also roll back to their respective checkpoints established before receiving these messages. This can cause a cascade of rollback activities, a phenomenon referred to as the domino effect. The domino effect can be avoided if the way that processes interact is controlled. One way is to restrict process interaction to accessing shared objects within the context of a transaction and the appropriate concurrency control and commit protocols.

3.2.2.2 Careful Replacement

A key issue in the development of reliable systems is the saving of consistent system states. The state of a system evolves in a number of volatile main memory pages. At some point in time, a consistent state is to be saved on non-volatile storage. The problem arises as to how to update the version of the system state on the non-volatile storage in such a way that if a crash was to occur in the midst of the update there would be a consistent system state available on the non-volatile storage when recovery begins.

The main issue is how pages on volatile storage are mapped to pages on non-volatile storage. There are two possible mappings -- direct and indirect. In direct mapping there is a one-to-one relationship between volatile and non-volatile storage pages. Objects are updated "in place." If a crash occurs in the midst of an update, an inconsistent state may exist. Indirect mapping uses techniques that aviod a one-to-one relationship.

The careful replacement technique updates a copy of the original object. The original copy, also called the "shadow" copy, remains unaffected in case of failures during the updating procedure. Only on commitment is the shadow copy replaced by the updated copy.

An example of this technique is the scheme proposed by Lampson and Sturgis [LAMP81] for making page write operations atomic in order to implement a stable storage facility. The Put and Get operations on a physical disk are not atomic in the sense that a crash of the system during the put operation for a page may leave that page only partially updated. A CarefulPut operation is defined to ensure that a put operation completes successfully provided no processor or disk crash occurs. A CarefulPut operation repeatedly writes a page and reads it until either it puts a clean page or some prescribed bound is exceeded. Similarly, a careful get operation reads a page repeatedly until either it gets a clean page or some prescribed bound is exceeded.

A Cleanup operation periodically checks the status of two pages; if one of the pages is corrupted and the other page is in good state, then the cleanup procedure replaces the contents of the corrupted page by the contents of the good page. This operation is periodically applied to each StablePage in the system. If Tc is the period of invoking the cleanup procedure, then for a StablePage to be reliable and highly available the period Tc must be small enough so that the probability of both DiskPages of a StablePage getting corrupted is infinitesimally small.

A StablePage is constructed from two disk-pages by procedures that use the CarefulGet and CarefulPut operations. A StablePut operation writes a StabePage by calling CarefulPut to write a main memory page to a disk page once and then calling CarefulPut write the same main memory page to a different disk page. StableGet is defined similarly by using CarefulGet.

Another example of careful replacement is the use of a shadow copy of an object that consists of multiple pages to facilitate recovery. A current version and shadow version of the object are maintained. The updates from an uncommitted transaction affect only the current version of the object. On transaction commitment, the current version is made the shadow version, thereby making the updates permanent. On transaction abort, the current version is deleted and the shadow version is made the most current version. The operation of replacing the shadow version by the current version must be atomic and done in one instruction. The technique described below [LORI77] does this.

Suppose that an object is represented by a set of StablePages {P1,...Pn} in the stable storage. The pages of the object are mapped from main memory to disk via a page table that has one entry per page. The old version of the object is preserved in a shadow page table that points to the pages of the shadow version. The current page table is initially set to the shadow page table to facilitate reading the object. Updates to pages of the object are noted in the current page table. When the current version of the object is to be written to stable storage, any pages of the object that have been updated are written to new disk pages using StablePut, the current page table is updated to show the mapping, and then the current page table is written to disk using StablePut. Any crash during the execution of this procedure, but before the completion of the last StablePut operation will abort the transaction. Successful completion of the last StablePut operation implies permanence of the updates.

3.2.2.3 Logs/Audit Trail

In this technique, actions performed on an object are recorded in a log or audit trail. The purpose of the logs is to support either undo of the logged action for state rollback, or redo the logged action to ensure permanence of results produced by committed transactions. Logs/audit-trails are used to either restore an object to a state prior to executing a sequence of operations on it or to ensure the permanence of the effect of executing a sequence of operations on it. The logs that facilitate object state recovery record the undo operation corresponding to every action performed on an object, and the logs that are used to ensure permanence of effect record the redo operation for every operation performed on the object. An undo record for an operation on an object specifies the actions to be executed to nullify the effect of executing that operation on the object. A redo record for an operation basically records the actions performed by the operation.

Logs that contain the redo actions are called the forward logs, and the logs that record the undo actions are called the backward logs. The backward logs either record the inverse operations or the values of the object before the application of the logged action. During a recovery process, a backward log is used by scanning it backwards for undoing actions in a last-in, first-out fashion. Thus a backward log can be viewed as a push-down stack. During system recovery, a forward log is scanned in the FIFO order as a queue.

A forward log is said to be idempotent if any number of (complete or aborted) repeated executions of the log from the beginning leave the updated objects in the same state. Such logs are also referred to as intention lists. One way to implement forward logs is to use differential files. In this technique, all updates to an object are recorded on a differential file. The updates from the differential file are periodically merged into the main copy of the object and such updates are then deleted from the differential file. The differential file technique provides a relatively inexpensive means of maintaining multiple versions of a large object. Intentions lists and forward logs are forms of differential files containing redo actions that record the new values of the objects and have the property of idempotency. The property of idempotency implies that repeated executions (some of which may be incomplete) of this sequence of actions would always bring the updated object to the same state.

The backward log technique is used when changes are made in-place in the stable storage. The recovery techniques based on backward logs follow the write-ahead-rule: (1) Before performing an operation in-place on an object, record the corresponding UNDO action in the log and force the log on the stable storage; (2) Before committing a transaction (i.e., sending a commit response to the user), either the updated versions of the objects or the corresponding forward logs must be forced on the stable storage. This rule makes sure that if the system crashes or the transaction aborts, the backward log can provide a means for restoring the object which has been updated in-place. Similarly, for a committed transaction, the updates made by it are guaranteed to be made permanent by using the forward logs.

3.2.2.4 Commit Protocols and Atomic Actions

Commit protocols are used for implementing atomic actions in a distributed system. The commit protocols enforce the atomicity of transactions in the presence of node crashes and communication link failures. The concept of commit protocols was independently introduced by Gray [GRAY79], and Lampson and Sturgis [LAMP76].

A transaction begins execution at a single node. When an operation is to be performed on objects at remote nodes a worker process, or cohort, is initiated at that node. When the operations of the transaction have been executed the processes execute a commit protocol to ensure that either all of the processes decide to commit or to abort the transaction. If the transaction commits, the updates are made permanent; otherwise, the objects are released in the state that they were in prior to the transaction's execution. This maintains database consistency by ensuring the "all-or-nothing" property of the global transaction.

The design of a commit protocol must address a number of issues. A decision must be made as to whether the control of the commit protocol is to be centralized or decentralized. If it is centralized, how a commit coordinator is determined must be defined. If it is decentralized, an efficient solution that minimizes messages must be devised. For both cases, what actions are taken if a failure occurs can impact system integrity and performance. If a failure occurs, a commit protocol could either cause all further access to an object to be blocked or not blocked. Ideally, the period of time that an object is in a locked state that is vulnerable to a failure should be minimized. Some of the desirable characteristics for a commit protocol are: 1) guaranteed transaction atomicity, 2) minimal overhead in terms of log writes, 3) optimized performance in no-failure case, 4) ability to "forget" the outcome of commit processing after a while, and 5) exploitation of read-only transactions [MOHA83].

A number of commit protocols have been proposed. The most common are one-phase and two-phase. They are both centralized, blocking protocols. The major difference between them is the length of time during which a cohort is vulnerable to the failure of a coordinator. How these and other commit protocols address the above mentioned issues are discussed in detail in the system designers guidebook.

3.2.2.5 Replication Management in Distributed Systems

A distributed computer system can offer benefits if objects are replicated and their management adjusted to take advantage of the multiple copies. The benefits can include improvements in performance and reliability.

The former is possible due to the reduction in communication cost to access an object and the increase in parallelism of operations on an object. The latter is possible because operations can continue despite the loss of system components. For example, if a directory is replicated on every site on a distributed system, the cost of reading it is the cost of accessing a local storage device (e.g., there is no overhead incurred due to communication between two sites). It is possible for users on multiple sites to be simultaneously accessing the directory, further improving a system's performance. Finally, if a site fails, the directory can still be accessed by any operating sites.

Unfortunately, increases in reliability and performance do not come for free and in many cases are not mutually attainable. This tradeoff in system attributes is often determined by a correctness criterion that describes a relationship between the values of the replicas of a distributed object at any point in time. The correctness criteria must ensure that a replication update algorithm satisfies the mutual consistency property: all replicas of an object converge to the same state and become identical if update operations cease.

The most common requirement of consistency has been based on the notion of serializability of transactions -- the effect of the execution of a set of transactions is equivalent to some serial schedule. This is called a strong consistency requirement. It requires that some subset of the set of copies of an object converge to a common state within the time it takes for a single transaction's execution.

A different requirement for consistency may be derived from observing applications such as directories, calendars, or network resource tables. The use of these objects does not require that they have the most up-to-date information. For example, a network name server may access an object's old site and be directed to its new site, or a message may be routed through a network over a longer than optimal path because its routing table is slightly out of date. But the services may be achieved with using non-identical copies of an object. The consistency requirement for them is that they must eventually converge to a common state if changes to the object stop. This convergence may span the time it takes for multiple transactions to execute. This correctness criteria is called weak consistency. It is a property of the application.

A third correctness criteria related to consistency exists. It is called semantic consistency and is a property of a set of transactions of an application. Semantic consistency seeks to find relationships (e.g., commutative, inverses, etc). between the effects of transactions that allow them to be executed according to a non-serializable schedule. To the best of our knowledge, semantic consistency has not been applied to performing updates on replicated objects. However, it has been proposed as a technique for merging replicated objects that existed in different network partitions when the partition is repaired.

In some sense, consistency criteria can be seen as points on a spectrum differentiated by the amount and type of activity that may occur in a system at any point in time. The three consistency criteria discussed are points in this spectrum that are currently known and are not meant to be interpreted as the only possible criteria.

The problem of managing replicated objects can be divided into four parts -- normal operation, detecting a failure and transitioning into a degraded mode of operation, operating in a degraded mode, and merging partitions during recovery. The first and third part of the problem are the same problem but in a different operating environment. They are almost always addressed by a single mechanism and will be discussed as a single problem in this paper. Transitioning into a degraded mode has two subparts -- termination and recovery. Termination is the action taken by operational sites when they determine that a site has failed and effects a transaction. Recovery is the action taken by a site to clean up any existing, uncompleted transactions when it becomes operational after previously failing. Finally, merging is when a set of sites acts to bring multiple copies of an object into a consistent state. It is helpful to recognize these distinct parts in order to understand the advantages, disadvantages, and applicability of the algorithms to be discussed.

A number of algorithms have been designed to ensure that the copies of an object meet some consistency correctness criteria. The system designers guidebook discusses how some of these algorithms operate and their effect under normal and degraded operation. Degraded operation exists when either a site goes down, a communication link is lost, a network is partitioned, or a message is lost or duplicated. Some update algorithms are tolerant of some of these failures and have no explicit distinction between normal and degraded operation. Other algorithms cannot tolerate failures and may block an operation until recovery from the failure has been completed, or abort the operation.

An attribute of interest is availability: the probability that an object can be accessed and an operation successfully performed. Those algorithms that ensure weak consistency result in a higher availability of objects. Strong consistency requirements typically restrict the concurrency level to a single update transaction and multiple read only transactions. They further restrict access to the replicated object by only one partition during degraded operation.

There are some general relationships among a replication algorithm's distribution of control, consistency criteria, reliability, and performance. Centralized control supports strong consistency and freedom from deadlock well, but is susceptible to single points of failure. It potentially can create performance problems (bottlenecks) and thereby reduce the availability of an object under both normal and degraded operation. Decentralized control can potentially increase the throughput of a system and the tolerance of a system to single point failures. Weak consistency is not appropriate for centralized control; it is naturally achieved through decentralized control. Weak consistency increases a site's throughput and response time, an object's availability, and a system's resilience to multiple failures.

3.2.2.6 Network Partitioning and Continued Operations

Under the conditions of network partitioning, allowing sites to update a replicated database, some copies of which are in an inaccessible partition, may result in inconsistency among the copies. This inconsistency among the copies requires resolving when the partition is repaired. In [BLAU83] two schemes, called Data Patch and Log Transformation, have been proposed for integrating the inconsistent copies of the database. The technique called Data Patch [GARC83a] relies on the data values and the semantic knowledge of the database. The technique of Log Transformation uses the logs of the transactions executed during network partitioning for the integration purpose.

Data Patch is an example of the forward error recovery technique. In this approach the data values before the partition and the data values at different sites after the partition are examined during the partition repair time. Depending on various different criteria and consistency requirements, the final merged value of the data is determined. The criteria and techniques for determining the repaired values are determined at the time of the database design; the database administrator uses tools based on these policies to integrate different copies of the database during the partition repair.

The usefulness of the data-patch technique strongly depends on a thorough understanding of the application environment. This technique fails to deal with network partitioning during integration. Data-patch allows ad hoc updates, but such updates require restrictions to keep the integration rules appropriate. As new transaction types are added, the integration rules must be updated appropriately. The compensating actions that require only the database values at the merge time are efficient to execute compared to those actions which require examination of the execution logs to determine which transactions generated these data values.

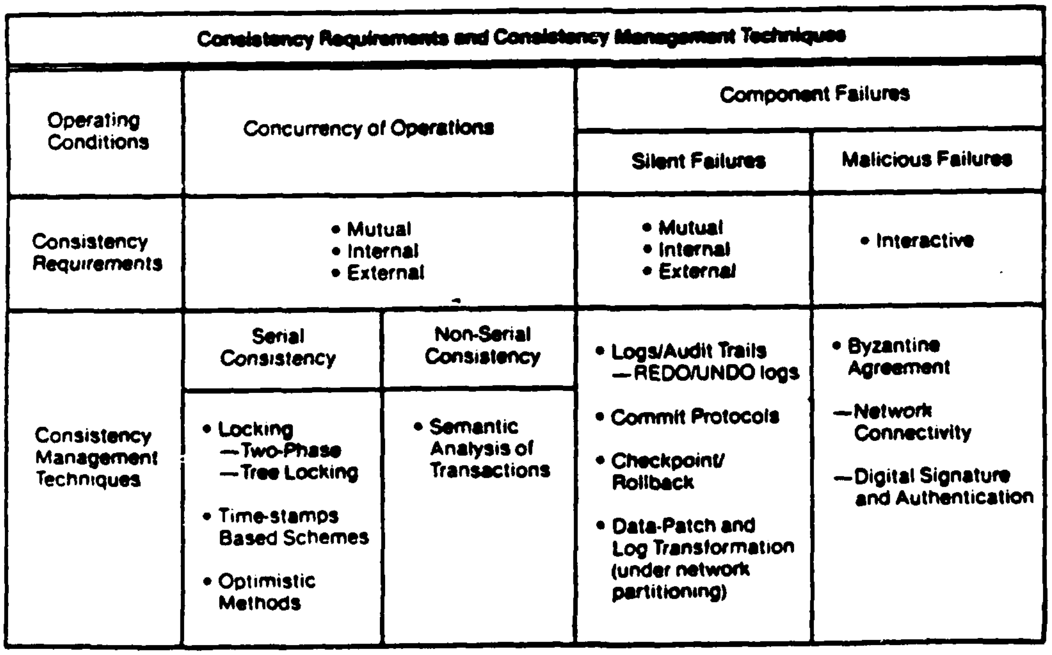
Log transformation relies on the logs of transaction executions at different sites for merging the partitioned copies of a database. This technique does not make use of the data values at the merge time. It assumes that all transactions are pre-defined and it requires that a database administrator specify the semantic properties and relationships between the transaction types. For example, transaction T1 and T2 commute, or transaction T2 overwrites all data written by T1. The transaction logs are merged according to these rules and other rules that apply integrity constraint checks.

During the partition merge time, the execution logs from each partition are exchanged and new merge logs are built. In constructing merge logs some conflicting transactions are undone and re-run. The merge logs are generated independently at each site; therefore, they may be different at different sites. However, the log transformation technique assumes that there is a system-wide policy to re-order conflicting transactions. One possible criterion for determining the order of execution for transactions in the merged logs is their execution time.

The applicability and usefulness of the log-transformation technique is dependent on the application. As in case of data-patch, the transactions in the system are of pre-defined types. As new transaction types are added to the system, necessary information is required to support correct operations of this technique.

3.3 Summary

In distributed systems, the operating conditions that may arise due to concurrency and component failures strongly influence the consistency management techniques. This is shown in Figure 3-1. Concurrency of operations requires techniques to maintain mutual, internal, and external consistency requirements. Depending on these consistency requirements, serializability of transactions may be a necessary requirement for the consistency management techniques; therefore the consistency requirements have been further divided into two classes: those that require serializability as a necessary requirement, and those that do not require serializability. The concurrency control techniques to ensure serializability are based on locking, time-stamp, or optimistic protocols. Most of the work in the area of consistency management has been in the context of maintaining serial consistency of distributed, replicated or partitioned databases. Not many researchers have addressed the consistency requirements that permit non-serializable interleaved executions of transactions. Applications with such consistency requirements can be important if continued operations are to be permitted in spite of network partitioning.



Consistency Requirements and Consistency Management Techniques

Operating Conditions	Concurrency of Operations		Component Failures	
			Silent Failures	Malicious Failures
Consistency Requirements	• Mutual • Internal • External		• Mutual • Internal • External	• Interactive
Consistency Management Techniques	Serial Consistency	Non-Serial Consistency	• Logs/Audit Trails —REDO/UNDO logs • Commit Protocols • Checkpoint/ Rollback • Data-Patch and Log Transformation (under network partitioning)	• Byzantine Agreement —Network Connectivity —Digital Signature and Authentication
	• Locking —Two-Phase —Tree Locking • Time-stamps Based Schemes • Optimistic Methods	• Semantic Analysis of Transactions		

Figure 3-1

The component failures have been divided into two classes: silent failures and malicious failures. Silent failure of a component means that the failed component does not generate or forward any information. In a malicious failure, the failed component may generate wrong messages or distort the messages it forwards. Silent failures of components affect the internal, mutual, and external consistency in the system. The techniques for maintaining system consistency under such failures are based on the concept of atomic actions. The problem of interactive consistency arises in the presence of malicious failures of components. The solutions to such problems are based on the solution to the Byzantine Generals' problem.

CHAPTER 4

ABSTRACT DISTRIBUTED SYSTEM ARCHITECTURE

The architectural features of distributed systems offer a great potential for designing reliable systems because the physical isolation between system components can reduce the correlation among component failures, and the redundancy of resources can support continued operations in the event of failures. Reliability and consistency management techniques provide the building blocks from which reliable distributed systems are built. However, this potential has largely remained unexploited because of the lack of a formal discipline to integrate the existing and known recovery techniques into the designs of distributed systems. This chapter presents an object-oriented design model for distributed systems which facilitates a systematic and well-structured integration of known recovery and consistency management techniques into the designs of distributed systems. We first discuss some of the techniques that can be used for structuring systems. Next a design model for reliable distributed systems is presented with a discussion of how the reliability and consistency management techniques described in Chapter 3 can be used to implement the functions of the design model. Finally, a system structure that combines the design model with object oriented design techniques is presented.

4.1 System Structuring Concepts

Much of the recent research in reliable system design is actually exploration into system structuring techniques. distributed systems are intrinsically more complex than centralized systems. A structured approach reduces design complexity. by factoring the designs into layers that create different levels of functional abstraction; the design of each layer can then be carried out somewhat independently of the design of the other layers. The layers in the system can be viewed as creating horizontal partitions in the system design.

Another structuring concept, which is dual as well as orthogonal to a layered approach, is object-orientation which creates vertical partitions in the system. The interactions between these partitions occur through some well-defined interfaces; thus, each partition in the system represents an independent domain where the internal structure of a domain can not be directly accessed by other domains. A vertical partition essentially embodies the concept of objects in the system. The whole system is viewed as a collection of objects. All state transformations in one partition by other partitions are performed through the interfaces defined by the partition. The advantage of such an approach is that the design of the internal structure of any given partition is independent of the designs of other partitions. These are the fundamental principles of data abstraction. From the viewpoint of reliable system design, such an approach is very attractive because it supports confinement of errors within an object boundary. This also implies that the recovery mechansims for a given partition can be designed to suit its reliability requirements.

There are two distinct approaches to designing reliable systems. The traditional approach takes a process-oriented view of the system where objects are bound to the address space of a process at the time of process creation and execution. The process is responsible for maintaining the integrity of these objects in the presence of faults and system crashes, and for recovering its locus of execution in the presence of faults. This approach uses checkpointing and rollback as primary recovery mechanisms for constructing resilient processes. Most previous operating systems have used system-wide checkpointing, saving the state of all processes in the system, irrespective of need. The research in this area has addressed the problems of separately checkpointing interacting concurrent processes [KIM79] [RUSS80]. The major problem is to avoid a domino effect in which the rollback of one process may lead to a cascade of rollbacks.

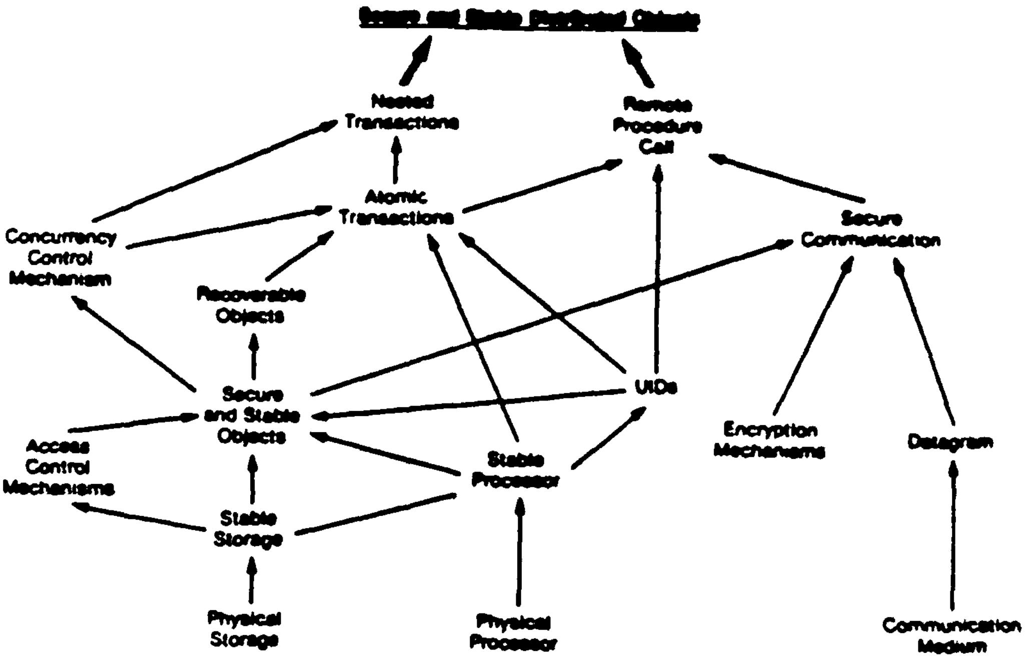
A second, more recent, approach, takes an object-oriented view of systems [LISK82]. In this view, objects are of distinct types; each type provides a defined set of externally visible operations. Each object is permanently bound to the address space of its object manager. Processes act upon these objects by invoking the visible operations implemented by an object manager. The object manager is responsible for enforcing necessary concurrency control rules and recovering objects from faults and system crashes. The primary recovery mechanisms include forward/backward logs, careful replacement, and object replication [KOHL81]. Processes are no longer responsible for recovering the objects they access during their execution; however, they are still responsible for recovering their execution locus. This requires establishing recovery points, and rolling back a process to some recovery point. A major advantage of the object-oriented approach is the clean separation between the recovery functions for processes and objects. Another advantage is that, for each object type, the recovery mechanisms and their design parameters can be selected to match the type's integrity requirements.

4.2 A Design Model For Reliable Distributed Systems

A design model for the construction of reliable distributed systems is shown in Figure 4-1. This model has been inspired by Lampson's lattice model [LAMP81a] for reliable distributed systems. The objective of reliable distributed system designs is to synthesize secure and stable distributed objects that survive crashes of system components and support high availability of functions. Such objects are constructed using unreliable resources such as physical storage (disks), physical processors, and the communication media. This section describes the design model in a bottom-up fashion. The properties of each functional abstraction in the model and the recovery mechanisms that can achieve these properties are identified.

The physical storage refers to non-volatile disk storage that has a non-zero probability of information loss. For example, a page on a disk may get corrupted due to a head-crash or some other malfunction. Another problem with the physical storage is the non-atomicity of write operations on pages. For example, a crash may occur in the disk system during writing a new value on a page. This leaves the page in an undefined state because the old value has been destroyed and the new value has not been written completely.

The stable storage facility, which is constructed from the physical disk storage, provides atomic write operations on pages. It enhances the availability of data by increasing the mean-time-to-failure of a disk page. Lampson introduced a technique for constructing stable storage from unreliable disc storage facility [LAMP81a]. It is based on careful replacement.



A Design Model for Reliable Distributed Systems

FIGURE 4-1

A physical processor loses its control state data on crashes; a restart operation can only cause a process to execute from the beginning. A stable processor facility, on the other hand, supports saving of process states on the stable storage, and restarting a process from some previously saved process state. The operation of saving process states is called checkpointing. Processes are considered as objects that are supported by a stable processor facility.

The next level of abstraction provides secure and stable objects based on stable storage, stable processor and unique identifier (UID) facilities. Stable objects are those that survive system crashes with a high probability and for which the primitive operations (i.e., the operations supported by the type definition) are atomic. Secure objects are protected objects which can only be accessed by authorized users.

Every object in the system is given a globally unique name using the UID facility. This name is never reused in the entire life-time of the system. From this unique identifier, the type of the object can be inferred. The UID also contains the identification of the node where the object was created. Objects in the system may migrate from one node to another. The UID facility defines the logical name space in the system. Operations on an object are invoked by specifying the UID of the object and the operation name. Because the type of an object can be determined from the unique identifier of that object, operation invocation on an object is directed to the appropriate object manager for that type. The operations on the remote and the local objects are invoked in an identical fashion. It is for this reason that we find the remote procedure call paradigm a convenient abstraction.

The UID generation is based on the stable storage and the stable processor facilities. The UID generation facility is based on a local clock process or a sequence counter that uses the stable storage to survive system crashes and to ensure that the same UID is not regenerated on restart of a node after a crash. The UID for an object indicates the type of the object and the node where it was created. A scheme for generating UID in a reliable fashion is described in [SCHA83].

The abstraction of recoverable objects provides mechanisms to restore the state of an object after having made some changes to it, or to commit a change to the object state. The concept of commitment forbids any restoration to states before commitment. Commitment of a change to an object essentially implies permanence of the changes made to the object since the last commit operation on it.

We use the concept of immutable versions to implement mutable recoverable objects. An immutable object is one that is never changed once it is created, i.e., every change to an object creates a new object. In our model every change to an object creates a new version of that object; this version is uniquely identifiable by using the UID of the object and the version number. These principles are discussed in [REED78] and [SVOB81].

Reliability techniques most suitable for constructing recoverable objects include multiple versions, differential files, intention lists, audit trails/logs, and self-identifying objects. Generally, a combination of several of these techniques is used in constructing recoverable objects at a node.

Maintaining multiple versions as a forward log is less expensive than as copies of the original object. A forward log in which the sequence of changes is idempotent can be used as an intention list to ensure the permanence of results on the commitment of a transaction. Backward logs are used for restoring objects by undoing the actions recorded in the log. Whenever a new uncommitted state of an object is to be forced in-place on the stable storage from the volatile memory, it is essential that (in order to keep the object recoverable) the backward log be forced on the stable storage before forcing the uncommitted object in-place on the stable storage.

Self-identifying objects and consistency checks play an important role during restart after a crash in reconstructing objects, object headers and directories during the restart after a crash. For example, with multiple versions additional information such as the object UID, state of the versions (committed, uncommitted, commit pending, etc.), pointers to other versions is incorporated for crash recovery. After reconstructing the data structures on crash recovery, the consistency checks are important in checking the validity and correctness of the reconstructed data structures.

Atomic transactions are implemented using the facilities described above and some concurrency control mechanisms. A transaction should be atomic in the presence of concurrent operations and system crashes. Atomicity of concurrent transactions requires suitable mechanisms for concurrency control. There are basically three distinct approaches to concurrency control: locking protocols [ESWA76], time-stamp based schemes [BERN81], and optimistic techniques [KUNG81]. Recoverable objects support schemes to achieve atomicity of transactions in the presence of system crashes. Transactions in our model are treated as objects of process type. As in the case of any other object in the system, a transaction is assigned a UID.

Nested transactions provide the facility to construct higher levels of abstractions by composing a set of already defined transactions into one larger transaction. The commitment of computations by each of the nested transactions is dependent on the commitment of the parent transaction. Concurrency control mechanisms are required to synchronize nested transactions of the same or different parent transactions.

The remote procedure call mechanism is based on an atomic transaction facility to ensure the atomicity of operations in the presence of system crashes and other concurrent transactions. The remote procedure call mechanism uses an unreliable datagram facility that supports high probability of successful delivery of messages. In [SHRI82a] and [LISK82a] arguments are given in favor of building a reliable remote procedure call facility using less sophisticated facilities such as a datagram. These are examples of end-to-end arguments [SALT81] that point out the wasteful duplication of functions at different levels. Secure communication is achieved by encryption of messages and storing unencrypted messages in protected buffers.

Table 4-1 summarizes and restates the design model in terms of design levels for a system. For each level the faults that can be handled, how they can be detected, and what error recovery techniques can be used are listed.

Table 4-1. Summary of Application of Recovery Mechanisms

Design Level	Fault Handled At This Level	Error Detection Techniques	Error Recovery Techniques
Distributed Applications and Distributed Objects	o Site Crashes o Link Failures o Lost Messages o Loss of Objects (Processes and Data) o Network Partitioning o Software Malfunctioning	o Time-out o Status Query o Consistency Checks o Acceptance Tests o Diagnostic Tests	o Object Replication - Majority Voting - Quorum Based Voting Schemes - Survivable Set o Primary/Stand-by - Periodic Checkpointing - Reconfiguration o Restart and Retry o Recovery Blocks - Primary/Alternate Blocks - Acceptance Tests o Exception Handling o Salvation Programs
Atomic Transactions	o Site Crashes o Memory Failures o Software Malfunctioning o Duplicate Messages	o Time-out o Status Query o Interface Test o Acceptance Test	o Commit Protocols o Conditional commitment of nested transactions
Recoverable Objects	o Site Crashes o Memory Failures o Software Malfunctioning o Loss of Objects (Processes and Data)	o Interface Tests o Acceptance Tests o Time-out o Periodic Consistency Checks	Backward Error Recovery o Multiple Versions o Differential Files o Intention Lists o Audit Trails/ Logs o Self-Identifying Objects o Salvation Programs o Incremental Dumping o Process Checkpoint

Table 4-1 (cont)

Design Level	Fault Handled At This Level	Error Detection Techniques	Error Recovery Techniques
Communication Level (Messages)	o Link Failures o Message Corruption o Lost Messages o Duplicate Messages	o Time-out o Status Query o Acks o Checksum/Parity Checks o Seq. Numbers	o Retransmissions o Alternate Links and Communication Paths o Replicated Messages
Disk Pages	o Read/Write Errors o Loss of Disks o Corruption of Pages	o Periodic Consistency Checks o Checksum/Parity Checks	o Careful Writes on Pages o Replication of Disk Pages o Pages Replicated on Multiple Disks

Techniques dealing with network partitioning, acceptance tests, interface tests, consistency checks, and exception handling are highly dependent on the applications.

4.3 A Model Of An Object Oriented Reliable Distributed System

This section takes the reliability design model presented in the previous section and integrates it into an on object oriented design. The concept of object managers is the basis for system structuring. An object manager provides the encapsulation for a given type of objects; all objects of that type are accessed or updated via that object manager. In this model the construction of reliable distributed objects is based on an atomic transaction facility and a remote procedure call mechanism. This approach is summarized in Figure 4-2.

The lowest layer in this figure represents the kernel functions that execute at every host node of the distributed system. Above the kernel layer are the local object management functions such as storage management, access control, synchronization, and object recovery. This layer represents the functions that are associated with every object manager in the system; the functions at this level deal only with the centralized object management. The next layer provides facility of atomic transactions; thus, a sequence of operations can be performed on a set of objects in an atomic fashion. The remote procedure call mechanism facilitates operations on objects that are not local. We have adopted the remote procedure call mechanism because it provides a uniform way of accessing remote as well as local objects; thus, location of the object is transparent to the users during access or update operations. It is important to make the semantics of remote and local procedure calls identical in the presence of host crashes and communication link failures. In our design we have adopted the "at most once" execution semantics for remote procedure calls; thus, in the presence of duplicate messages or on server node crash-restart, effectively only one execution of the remote procedure will occur. The combination of the remote procedure call mechanism with the atomic transaction facility is used for managing objects that are either partitioned or replicated. Based on these mechanisms one can suitably create type definitions for replicated or partitioned objects such that one can access or update those objects in the same manner as updating centralized objects.

DISTRIBUTED OBJECT MANAGMENT FUNCTIONS
(Partitioned and Replicated Objects)

RELIABLE REMOTE PROCEDURE CALL MECHANISM

ATOMIC TRANSACTION FACILITY

LOCAL OBJECT MANAGEMENT FUNCTIONS
(Concurrency Control, Recovery, Access Control,
Object Storage Management)

KERNEL FUNCTIONS
(Host Resource Management, Communication, Scheduling,
Remote Call Handling, Interrupt Handling)

HARDWARE

A Model for Reliable Distributed systems
Figure 4-2

4.3.1 Structure of Object-Oriented Distributed Systems

An object-oriented system consists of a collection of Type Managers and the objects created by them. Type Managers create vertical partitions in the system. For a given type in the system, a Type Manager would exist at all those nodes which may be required to store objects of that type. A Type Manager at a node manages all objects of that type at that node. The multiple instances of Type Managers for a type function cooperatively to provide the abstraction of a single Type Manager for that type in the system. Each Type Manager defines an address space in which all the objects of that type reside. A Type Manager is logically viewed as a single process that performs all the state transformations on the objects in its address space in response to execution requests by some other objects of the same or different type.

At a physical node, several different Type Managers may reside, each managing objects of its type at that node. The abstract machine to support such an object-oriented system can be constructed from almost any hardware/software system architecture. The system architecture, which includes the hardware, software, and the firmware architecture, of the processors to support such a system must have: (i) a mechanism for switching the processor between Type Managers, (ii) a mechanism for partitioning secondary memory resources among Type Managers, and (iii) a mechanism for exchanging messages between Type Managers.

It can be seen from the preceding model of Type Managers that there is no concept of a system-wide state or uniform control and/or recovery mechanisms. Resource management functions and recovery mechanisms are partitioned along with the set of Type Managers. The traditional functions of system-wide software units such as operating systems and database systems are incorporated into a collection of Type Managers which implement the basic elements of the model of distributed computations. This is a radically new view of operating systems.

Object Type Managers are the primary building blocks for the permanent elements of the system. The Type-Type Manager is an object in the system that manages "types" in the system. It is the means by which new types are introduced into the system. The concept of the Type-Type Manager is essentially the same as that of the TYPE-TYPE object in the Hydra design [COHE75]

The objects in the system are accessed in a uniform fashion regardless of their locations. All operations on permanent objects are performed within a transaction. A transaction is basically an atomic action that is defined as a sequence of operations on local or remote objects. A transaction ensures atomicity of distributed operations. It is possible to introduce concurrency within a transaction by creating one or more nested parallel transactions.

4.3.2 Functions of the Type Managers

The functional characteristics implemented by the Type Managers are the original basis for defining abstract data types. Extending abstract data type concepts to include a formal basis for the integration of recovery, synchronization, and access control mechanisms generates a number of additional functions for the Type Managers:

1. Each Type Manager is directly responsible for the mapping of the occurrences of the objects they define to physical storage.
2. Each Type Manager implements access control policies for the occurrences of its type.
3. Each Type Manager supports concurrent execution of its procedures and/or functions.
4. Each Type Manager ensures the consistency of the objects it stores under concurrent and distributed use.
5. Each Type Manager implements the necessary levels of redundancy to ensure the level of fault tolerance given in its specification.

This obviously integrates many functions that have been conventionally associated with database systems into the object management functions of this operating system.

4.3.3 Structure of Type Managers

Externally viewed, a Type Manager is a collection of functions and procedures which can be invoked on the objects of its type by specifying the identifier of the object along with the operation name. This causes an invocation request message to be sent to the Type Manager regardless of its physical location in the system. Internally, these operations are executed by the Type Manager using one or more server processes; such server processes may be dynamically created or destroyed by the Type Manager. The operations on remote and local objects are invoked by the clients in the same fashion as procedure call. Such invocations on remote objects are performed by implementing remote procedure calls [NELS81] [SHRI82] with "at most once execution" semantics. A Type Manager consists of:

- Data structures for the objects of that type;
- Procedures/functions defining the type;
- Concurrency protocols;
- Recovery mechanisms;
- A database to manage the objects in its domain;
- A controller process that schedules/executes the requests.

A Type Manager is responsible for the permanent storage of the object instances of its type. Each Type Manager interfaces directly with some set of permanent storage devices. The Type Manager generates the mapping from the UID for an object of its type to the physical storage on some permanent storage devices. It also realizes object instantiation in the executable volatile storage from the permanent storage. There is no system-wide file system. The object management system takes the place of a file system.

A Type Manager consists of a controller process whose purpose is to schedule server processes to serve client requests. The server process is given the same UID as that of the client process; thus, a client process is conceptually viewed as migrating into the address space of the Type Manager. This view of the migrating client process is useful from the viewpoint of enforcing access rights associated with the client process. On the completion of the requested service, the server process is deallocated. The controller process accepts the incoming or outgoing invocation request messages, performs security checks, and interfaces with the kernel procedures. Effectively, the controller process plays the part of a local operating system for the Type Manager; the scheduling policies can thus be tailored to the specific requirements of the Type Managers. The controller process manages the server processes performing the operations and provides them with a set of procedures that perform resource management, communication, protection and other services that are normally provided by an operating system.

A Type Manager's controller has several responsibilities related to protecting its objects from unauthorized access. Upon receiving an invocation request, the controller must obtain and store the requesting process' identification. This information is made available to the operation via a callable procedure so that the Type Manager's controller may check the access list of the object. In addition, the controller appends the identification of a process which is making an outgoing invocation request to some other Type Manager.

When an incoming invocation request is received, the controller attempts to locate the object whose UID is given in the request. First, the controller looks for the object in its own local pool of objects. If found, the program which will perform the operation on the object is parameterized with the object's local address and then is scheduled as the server process. If the object is not found locally, the controller determines if a "forwarding address" has been left for that object. This might occur if the object has been relocated to some other host. If the object is not found locally, the controller sends a reply message indicating that the object was not found and includes the forwarding address if any.

In response to an update request, the Type Manager creates a new version of the object. This version is committed only when the transaction that created it commits; the uncommitted versions are discarded if the transaction aborts.

Each Type Manager maintains a database which records the necessary information pertaining to the objects in its address space. This database records the identifiers of the objects of that type currently present at that node, their physical addresses, and the commitment status of their most current versions. A Type Manager is also responsible for aborting a new uncommitted version by timing out if it detects no activity of the transaction that created this version. Every time a new version of an object is created by a transaction by invoking an update operaticn, tne Type Manager ensures that this new version is written onto the stable storage before sending an acknowledgement for the operation. A scheme for maintaining such multiple versions using differential files is described in Chapter 4.

Type Managers are responsible for ensuring that each of their defined operations is atomic. The operation must either complete successfully or else abort, leaving the object completely unmodified. This is not difficult to achieve if only local objects are being modified in the operation. However, if the operation involves invoking operations on other Type Managers, then the controller uses the transaction facility to ensure the atomicity of the update. If the Type Manager is structured so that operations may be executed concurrently, the controller ensures that objects are not being modified by two operations simultaneously or read by one operation while being modified by another. Each type, in general, has its own set of constraints on the allowed order of execution of its operations on a given object. These constraints are supplied when the Type Manager is created.

4.3.4 Distributed Types

The reason for introducing the concept of distributed types in the system is to make transparent the distributed nature of an object that is logically viewed as a single object. The components of an object may be distributed by replication or partitioning. The transparency of the replicated or partitioned nature of an object is a convenient abstraction which makes updating and accessing of distributed and centralized objects identical.

A distributed type is an abstract data type whose concrete representation is distributed. For example, an abstract type called reliable-file might be implemented using physically distributed replicated copies of a file, or a global database might be implemented as a set of partitioned distributed components. The consistency and coordination among the distributed components of the concrete representation is specified in the type definition and enforced by the distributed Type Manager. Unlike the centralized objects, an occurrence of a distributed type does not have a unique host location, i.e., an object of a distributed type may "reside" at more than one host for reliability and performance reasons. An occurrence of a distributed type is given a UID, the Type Manager then maps the operations directed to this UID into a set of operations, which are executed as a transaction, on the components that comprise the distributed object's concrete representation. This mapping can be done at any of the hosts where the distributed object is conceptually "residing". The operations defined for a distributed type are implemented as transactions.

4.4 Summary

The system designers guidebook presents an object-oriented design model that supports structuring of distributed systems for high reliability and error recovery. In this model, we identify the error recovery problems at the different levels of functional abstraction and show how various error recovery techniques are integrated into this design model. For example, techniques based on multiple versions, logs, careful replacement, and differential files are used for constructing recoverable objects, checkpointing. Commitment techniques are used for constructing atomic transactions, and the techniques based on replication and primary-backup modes of operation are used for constructing reliable distributed objects. The use of this model in the design of an actual distributed operating system is the topic of the next chapter.

CHAPTER 5

ZEUS: AN EXAMPLE SYSTEM DESIGN

The previous chapter presented several recovery mechanisms and a design model for constructing reliable distributed systems. This design model provides a framework for integrating the recovery mechanisms into a system design in a structural fashion. Ideally, a distributed operating system should make the low level recovery mechanisms, such as logs and commit protocols, transparent to application programmers by providing some high-level functions for constructing reliable software. This chapter describes a distributed operating system called Zeus which has been designed with this in mind. The design illustrates how various recovery mechanisms are integrated according to the design model presented in the previous chapter.

This example design should be viewed as a framework for integrating recovery mechanisms into distributed system designs rather than a point solution. As mentioned earlier, the approach to the development of the system designers guidebook is example-driven. This approach consists of designing an example system which illustrates the structuring principles as well as the formal design definition methods for reliable system designs. Additionally, the same example design is used to illustrate the application of design analysis and verification method to reliable distributed systems to analyze their performance, reliability, and functional correctness.

This chapter presents the principles followed in designing Zeus, an object-oriented distributed operating system for integrating recovery mechanisms into the designs of distributed command and control systems. The main contribution of this work is an operating system design that provides an integrated set of functions to application programmers for reliable management of objects in distributed systems. These functions transparently provide complex recovery mechanisms, commit protocols, concurrency control mechanisms [KOHL81] [BERN81], and remote object accessing to application programmers. For now the primary goal of the Zeus design is to define reliable object management functions for distributed command and control systems and to evaluate the performance and the correctness of the recovery mechanisms for these functions; therefore, no implementation of the design exists at this stage. The user visible functions support definition of object types, creation of objects, and updating of distributed objects using atomic transactions.

A distributed operating system for high reliability applications must not only include suitable recovery mechanisms that are transparent to the application developers but it should also provide transparency of the distributed nature of the system. The second feature is important to make development of distributed software no more difficult than the development of conventional software systems. The Zeus design has made a significant contribution in this direction. Some other systems, such as LOCUS [WALK83], have integrated these two concepts in their designs; however, in most of these systems, object management is limited only to the file storage level. To date, Argus [LISK82] is the only other system besides Zeus which provides a set of general mechanisms for reliable management of distributed objects of any type. Zeus not only provides such general mechanisms, but also addresses several other issues not included in the Argus design such as object naming, object relocation, authentication and object protection. We have made an effort to address these issues in the Zeus design making it novel as compared to any other distributed operating systems. Another novel feature is the integration of the conventional database management functions into the operating system object management functions. This is an important advance in the operating system designs because most of the current popular operating systems do not provide efficient mechanisms for database applications [STON81]. Even with respect to its recovery model, the Zeus design differs significantly from other known designs.

The concept of object-oriented design has been used in some recent distributed system designs such as Cronus [SCHA83], SWALLOW [SVOB81], Argus [LISK82], and in the approach presented in [SHRI81]. Argus provides object-oriented linguistic mechanisms for constructing reliable distributed systems, and SWALLOW provides reliable object management. These systems do not support some of the other operating system functions such as access control, naming, sharing, and resource management. Some of the functions supported by Zeus, such as naming, authentication, and interprocess communication, can be found in Grapevine [BIRR82]. Grapevine can not be regarded as a general purpose distributed operating system because it is intended only to support a distributed mail system.

The design of the Cronus operating system has significantly influenced the design of Zeus, largely because both these systems are intended for highly reliable applications such as command and control systems. Zeus provides users with reliable object management, which is not present in the current design of the Cronus system. Like Cronus, Zeus has the character of a general purpose operating system mainly because the nature of the command and control applications includes a wide range of processing characteristics. This is in sharp contrast to the requirements for banking or airline reservation systems where the application environment is well-defined. Zeus provides capabilities for defining and creating objects and transactions required by the application systems. It also provides mechanisms that support management of such objects in a reliable fashion. Zeus can be used for constructing any high reliability application system.

This chapter presents the basic object-oriented building block mechanisms provided by the Zeus distributed operating system. The concept of object managers is the basis for system structuring. An object manager provides the encapsulation for a given type of object; all objects of that type are accessed or updated via that object manager. The object-oriented recovery model underlying the Zeus design is described in Chapter 4. In this model the construction of reliable distributed objects is based on an atomic transaction facility and a remote procedure call mechanism.

The object management model used in the Zeus design is based on the concepts developed in the Hydra [COHE75] design. There are some obvious differences between the protection models used in the Hydra and Zeus designs. The protec:ion mechanism in the Zeus design is based on access control lists while the Hydra model is capability based. Although both these models are equivalent in terms of their functionality, they differ with respect to their operational environment. The prime reason for using the access control list model in our design is to be able to change the access rights dynamically. Although it is not very efficient to change access rights dynamically in a capability based system, it is important in a command control system where some of the nodes might be taken over by hostile forces.

5.1 Structure of the Zeus System

Zeus is essentially a collection of Type Managers (TMs); typically, many different Type Managers coexist on a host node. The core of the operating system consists of a set of Type Managers that support capabilities for defining new types and object instances in the system, authentication of users, naming environment for each user, and reliable process and transaction management functions. These system-defined Type Managers reside at every node in the system. The lowest level of operating system at each node is called the kernel; the kernel virtualizes the resources at the host so that each Type Manager can be viewed as having its own virtual processor. The kernel supports interprocess communication, primary storage management, processor scheduling, interfaces to secondary storage devices, and UID generation; it also handles all interrupts due to storage devices and the communication devices. Figure 5-1 shows the major components of the Zeus system.

5.1.1 Structure of the Zeus Kernel

The Zeus kernel provides low level services to the Type Managers of the system. These services include three important functions 1) interprocess communication, 2) storage management and 3) unique identifier (UID) generation. The UID generation in turn depends on the failure detection and recovery of hosts in the Zeus system. The kernel consists of a task dispatcher and a number of interrupt handlers. The task dispatcher schedules the different Type Managers at its host node and handles their requests for resources. It also handles the restart of the system and initiation of the Type Managers. The resources managed by the kernel include volatile and non-volatile storage, the processor and the communication handler. The kernel interface consists primarily of three parts: invocation requests to other Type Managers, requests for unique numbers, and requests for resources.

Interprocess communication is achieved by the mechanism of remote procedure call (RPC) which consists of four messages interchanged between caller and callee. These are call, call acknowledge, response and response acknowledge. For each call that is made from or to a Type Manager the status of the call parameters and status must be stored. To do this each Type Manager has a call handler to perform this function. The synchronous nature of the RPC is achieved by the Type Managers who will first issue a call and then, on getting the response, will inform the caller of it.

The storage functions of the kernel are performed at the object level; thus, calls to the kernel can retrieve, store and delete objects. Further stable storage operations can be executed by the kernel, where stable storage is implemented using the Lampson [LAMP81] scheme. Storage management in the kernel is minimal. Storage is available in fixed sized blocks and the Type Managers request one or more of these blocks at any time. A Type Manager is solely responsible for the data it writes to the blocks of storage. The kernel keeps track of the ownership of blocks of storage. The routing of invocation requests to Type Managers is the major function of the kernel. Each call is an operation invoked against an object that is held by some Type Manager. Operation Switch, which is a component of the kernel, supports this function.

UID generation is a function used by the RPC and by the Type Managers so that calls and objects can be uniquely identified. This function must continue despite failure and recovery of hosts. To achieve this the hosts participate in a distributed computation to keep track of active hosts and to let new or recovered hosts join in the UID generation function.

The function of the Operation Switch is to forward an invocation request to the appropriate Type Manager at a local or a remote node. These calls may be from a Type Manager or from the network driver. Each call contains the following information:

1. The extended UID of the object against which the call is invoked.
2. The extended UID of the process invoking the operation.
3. The extended UID of the principal on whose behalf the operation is being invoked.
4. The operation and a set of parameters.

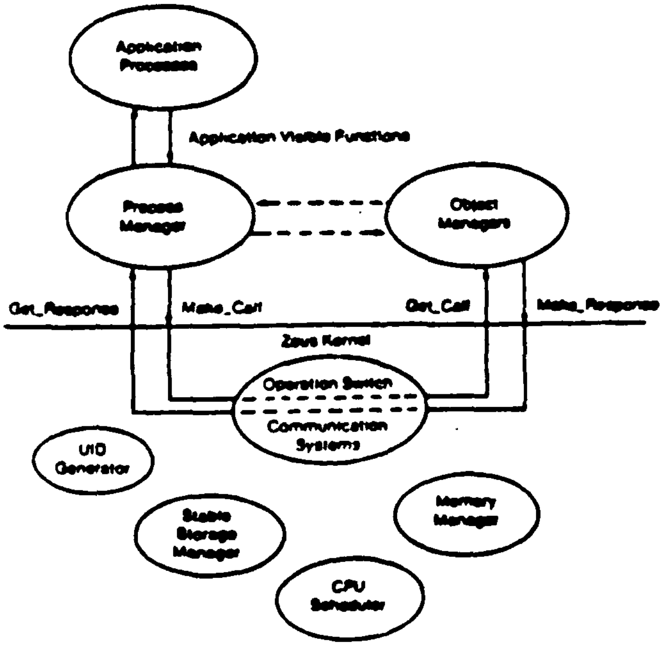


Figure 5-1. Overview of the System Architecture

The Operation Switch uses the host hint field of the target object's extended UID to determine whether the object is on the host or not. If it is, it uses the type unique number of the object to direct the call to the proper Type Manager. If the object is on another host, the Operation Switch instructs the Network Handler to send the call to the other host.

5.1.2 System-Defined Type Managers

As mentioned previously, Zeus is a set of Type Managers whose members may potentially change dynamically as Type Managers are created, deleted, and modified. There is, however, a subset of Type Managers called the System Type Manager which perform the essential services provided by the kernel of a conventional operating system. In this section, the Type Managers for these system types are defined. The following are the System Type Managers which exist at each node in the system:

(1) Type-Type Manager
(2) Process/Transaction Manager
(3) Principal and Authentication Manager
(4) Symbolic Name Manager
(5) Program Type Manager
(6) Message Type Manager

The definitions of new Type Managers is introduced in the system by using the mechanisms supported by a system-wide object called the Type-Type Manager; thus, the Type-Type Manager implements functions to create, alter, delete and replicate Type Managers. The definition of the Type-Type object given here is an adaptation and extension of the Type-Type concepts originating in the HYDRA [WULF81] operating system. The facilities provided by the Type-Type Manager include an explicit command for locating the copies of a Type Manager.

The Process/Transaction Manager provides the reliable management of processes and their operations in the system. The atomic action facility, called transaction, forms the basic mechanism for building reliable applications including management of distributed objects.

The Symbolic Name Manager and the Message Type Manager can be regarded as applications built using the Process Manager functions. The Symbolic Name Manager maintains the name contexts for the clients in the system. Thus, a client can use string names instead of UIDs for accessing objects; the Symbolic Name Manager translates the string names to object UIDs depending on the context of their use. The Message Type Manager supports message cmmunication among the clients. The Program Type Manager supports building executable program objects from a set of specified code segments. It has the conventional functions of a linker and loader. The Principal and Access Control Manager has the function of associating appropriate access rights with the processes in the system which carry out operations on behalf of system users.

5.1.3 Process Management

Processes are active objects that perform state changes on behalf of system users by modifying shared permanent objects. They have a (system defined) type, PROCESS, and are managed by an Type Manager called the Process Manager. Transactions are PROCESS objects with the additional property of atomicity. Atomicity, or the "all or nothing" property, means that either all or none of a transaction's updates become permanent. The TRANSACTION type is derived from the PROCESS type; thus, all operations defined on processes are applicable to transactions. Additional operations are defined for transaction objects. Shared objects can be updated reliably only by transactions.

Reliable applications are built in Zeus by manipulating objects using transactions. The Zeus kernel offers only unreliable remote procedure calls [NELS81] [LAMP81b] which are made reliable by invoking them within a transaction. The transaction facility also provides a powerful mechanism for managing replicated or partitioned objects reliably. This section presents the computational model for managing processes and transactions, and the application visible operations. These operations are summarized in Table 5-1.

The Zeus design uses the transaction concept for reliability and to avoid the domino effect during process rollback by enforcing disciplined interactions among processes. First, all information-flow among processes which affects global state takes place via shared objects. Second, all shared global objects must be accessed within a transaction. A transaction defines a "sphere of control" (SOC) [DAVI73]; all objects modified by a transaction are said to belong to its sphere of control. Third, no other process/transaction is allowed to access objects belonging to a transaction's sphere of control during that transaction's execution. If the transaction completes successfully, the updated objects are "committed"; otherwise, they are restored to their state before the transaction began execution.

A process can create sequential and concurrent transactions. The parent process of a sequential transaction is suspended until that transaction terminates. However, the parent process of a concurrent transaction process executes concurrently with its child. When a concurrent transaction process terminates, an appropriate condition is signaled to the parent process.

The Zeus design allows a transaction to invoke other (sequential and concurrent) transactions, called nested transactions [MOSS81] [RIES82]. A top-level transaction is one whose parent is a non-transaction type process. Zeus supports nested transactions (i) to introduce concurrency into an atomic action, and (ii) to allow a transaction to invoke procedures which may contain transactions. Nested transactions also provide a means for constructing recovery blocks [HORN74], and updating replicated objects using majority consensus [THOM79] or weighted voting [GIFF79].

Table 5-1: Application Visible Operations

OPERATION	REMARKS
INVOKE	input parameters: UID(1) of object to which operation is applied, operation name and operation parameters
CREATE_PROCESS	if successful, returns UID of the new process, otherwise, returns error signal
DELETE_PROCESS	input parameter: UID of process to be deleted
BEGIN_TRANSACTION	if successful, returns UID of the new sequential transaction; otherwise, returns error signal
CREATE_TRANSACTION	if successful, returns UID of the new concurrent transaction; otherwise, returns error signal
END_TRANSACTION	initiates commit protocol between Process Manager and Object Managers
WAIT	input parameters: transaction UID(s) on which parent, waits, optional timeout value
COMMIT	invoked by processes only; input parameters: transaction UID
ABORT	cancels all of a transaction's updates
ESTABLISH_RECOVERY_POINT	returns recovery point number
DISCARD_RECOVERY_POINT	input parameter: recovery point number
ROLLBACK	input parameter: recovery point number; without parameter, process rolls back to most recent recovery point

To create a sequential transaction, a parent process or transaction invokes the BEGIN_TRANSACTION function. The parent process is then suspended until its child terminates. The sequential transaction created by BEGIN_TRANSACTION inherits its parent's address space and runtime environment. The transaction terminates by executing either END_TRANSACTION or ABORT. Invoking END_TRANSACTION causes the Process Manager to execute commit protocols with the Type Managers (also called object managers) of the objects

(1) A UID is a globally Unique Identifier; every process, transaction and object in the system has a UID.

accessed by the transaction. The code between a BEGIN_TRANSACTION and a corresponding END_TRANSACTION is executed as an atomic action, that is, as a transaction.

A process or transaction creates a concurrent transaction by invoking CREATE_TRANSACTION. When a concurrent transaction completes, a condition is signalled to its parent. At this point the child transaction is still not committed; it is either in the aborted or the commit-pending state. The commit-pending state indicates that the transaction was successful and is waiting for its parent to issue a commit command. If the parent is a non-transaction process, then it explicitly issues the COMMIT command. Nested transactions are implicitly committed when the top-level transaction commits.

A parent process or transaction can wait for a completion signal from a concurrent child transaction by invoking the WAIT function. WAIT can include a time-out option will cause the invoker to be suspended until either the transaction completes or an interval of time passes. A process may wait on any of several transactions, or until each of a set of transactions has completed.

Processes and transactions perform operations on shared global objects using the INVOKE function. Remote and local shared global objects are accessed identically.

A top-level transaction may make a commit decision based on the status of its nested transactions (e.g., completed or aborted). It is undesirable to require a top-level transaction to revalidate the state of the objects accessed by a completed nested transaction if the top-level transaction decides to commit. Revalidation of object states can be avoided if a nested transaction follows an appropriate commit protocol. A nested transaction can follow either a one-phase or two-phase commit protocol [BALT81] with its parent. Using a one-phase commit protocol means that, when a nested transaction completes, all th_ objects it modified are in the commit-pending state. The commit-pending versions cannot be aborted unilaterally by an Type Manager. In contrast, following a two-phase commit protocol leaves modified objects in the uncommitted state. Such uncommitted versions can be aborted unilaterally by their Type Managers, thereby aborting that nested transaction.

Zeus uses the one-phase commit option for nested transactions. This allows the use, within a transaction, of a conditional statement that depends on the successful completion of one of the transaction's nested children. Such conditionals may be used because the one-phase commit option prevents a unilateral abort by an object manager from invalidating conditional decisions made by the parent transaction. It also eliminates the need for a parent transaction to revalidate the status of completed nested transactions.

Object managers and process managers follow a two-phase locking protocol [ESWA76] so that all concurrently executing transactions are serializable. All concurrently executing nested transaction with the same parent are also serializable according to these rules.

A process or transaction may establish a recovery point by invoking ESTABLISH_RECOVERY_POINT (ERP). When ERP is invoked, the Process Manager saves the current state of the process on stable storage and returns a recovery point number to the calling process. A process can explicitly roll back to some previous recovery point by invoking the ROLLBACK function. If no parameters are given, the calling process rolls back to its last recovery point. If a recovery point number is supplied, the process rolls back to that recovery point.

It is possible to establish a recovery point for a parent process when a sequential transaction commits by using the ERP option with the END_TRANSACTION command. If the parent process subsequently crashes, it would be started either from this recovery point or from a subsequent recovery point, avoiding re-execution of a transaction that has already been committed.

The Process Manager establishes the initial state of every process as the recovery point numbered 0. All subsequent calls to ERP return sequentially increasing integer numbers. When a process completes, all of its recovery points are discarded. A process can also discard any of its recovery points by invoking DISCARD_RECOVERY_POINT (DRP), with the number of the recovery points to be discarded.

5.2 Formal Definitions of the Designs

A major part of detailed design of the Zeus operating system that includes the design of the Process/Transaction Manager and the Generic Object Manager has been done using Concurrent System Definition Language (CSDL). These designs are presented in Volume II of the guidebook.

CSDL is intended for designing systems with inherent concurrency (for example, geographically distributed systems), systems in which concurrency is needed to deliver adequate performance, or for which expressing the design as a collection of concurrent modules leads to a simpler, more understandable design.

There are two basic concurrent architectures: the static architecture in which the system is created with a fixed number of modules which persist throughout its lifetime, and the dynamic architecture in which modules are created as needed to handle new tasks. CSDL supports them both. Following are the salient features of the CSDL methodology:

1. A formal model of sequential and concurrent computations.

2. A system model that characterizes the building blocks with which systems may be designed.

3. Methodological principles and guidelines that define desirable properties of the design activity, the design language and the design itself, and make procedural suggestions for carrying out the design process.

4. Technical methods essential for engineering software. They are, for example, data abstraction, procedural abstraction, Dijkstra's constructive approach, and the like.

5. A description language - a formal notation for describing how a system is built up from pieces and how those pieces are connected. Its semantics are based on the model of computation.

6. A specification language - a formal notation for documenting the expected behavior of a system description. Its semantics are based on the formal model.

7. Analytic methods for investigating operational properties such as performance, reliability, or security of alternative functionally correct system designs.

These elements are applied to detailed design, the development phase whose work product is a design documenting a system's logical architecture, its paths of information flow, the data type of each system object and the behavior of the system and each of its modules. A detailed design expresses what will actually be implemented. Each object in the design -- module, data object, procedure, or information flow path -- will exist in the implementation, though the object's physical realization may be different from its logical design. For example, a type operation designed as a procedure may be implemented with in-line code.

In CSDL, the basic locus of control is the machine. The machine is a container of objects and a control procedure in execution. A machine may contain data objects of any type. A machine may also contain machine-objects, that is, other machines in operation, and pools of machine-objects from which operating machines may be created and destroyed. These structures (the machine-object and the pool of machine-objects) enable a single machine to contain several concurrently operating local loci of control.

A machine definition consists of a list of the machine's public objects and specifications of the machine's externally visible behavior. Public objects are those (active and passive) machine objects which define the external view of the machine. A machine's realization is guaranteed to have these objects. A machine communicates with its environment through its active public objects. Its passive public objects are visible to the environment, but cannot be manipulated by it. Public objects are used in specifications of the machine's externally visible behavior. Machine specifications may specify initial values, invariant properties and machine behavior.

A concurrent system is a collection of machines which operate concurrently and autonomously. They communicate asynchronously by passing information. Internally, a machine consists of data objects and procedures and/or subordinate machines to manipulate these objects. A machine containing only procedures constitutes a sequential locus of control. A machine containing subordinate machines constitutes several autonomous control sites. If the system's architecture is viewed as a tree, its leaves are all sequential control sites.

Machines may also contain machine pools from which machine instances may be created and destroyed as the system runs.

Systems are evolutionary. The initial system configuration is described by a distinguished machine, SYSTEM. SYSTEM may contain other machines and machine pools. Each machine that SYSTEM contains may, in turn, contain other machines and machine pools. The initial system is, then, the configuration consisting of SYSTEM, all machines it contains, and all the machines they contain. A system evolves by dynamic creation and destruction of machines from pools. Since every pool element may contain machines and machine pools, creating a new machine dynamically may, in effect, create a new subsystem.

A system's communication architecture is the set of connections among its machines. Connections are formed among active objects, objects whose values can change without being manipulated by the machine which contains them. Since machines cannot manipulate each other's objects, a communication link is set up by connecting an active object in one machine to a complementary (roughly same type, opposite direction) active object in another. The sending machine puts a value in its local active object, and that value is instantaneously transmitted to the complementary active object from which the other machine can get it by a local operation. Active objects may be connected to realize point-to-point, multi-cast, fan-in and broadcast communication architectures. Connected active objects by definition correspond to shared objects in the computational model.

5.3 Summary

The object oriented design model presented in the previous chapter is used for designing Zeus, a distributed operating system for reliable applications. A Zeus system is essentially a collection of Type Managers; each Type Manager is responsible for managing the objects of its associated type. A set of system-defined Type Managers provides certain primitives for building reliable application systems. An atomic action facility is the basic mechanism in Zeus for building reliable applications. An atomic action in Zeus, called transaction, can span over several distributed sites in the system. The purpose of the Zeus design is to illustrate certain design principles for building reliable distributed systems; it is not intended as a point solution but rather a framework for system designs. In designing a system, the designer has to go through several steps starting with its conceptual design to the implementation. A formal notation which supports expression of the system definition in a clear and systematic fashion is the single most important tool for the designer. Concurrent System Definition Language (CSDL) is intended to serve as a formal design notation. A major part of the Zeus design was defined using CSDL.

CHAPTER 6

ANALYSIS AND VALIDATION TECHNIQUES

6.1 Introduction

Reliability, timeliness and correctness of system functions are the most critical attributes of a command and control system. A major part of the system designers guidebook is devoted to the techniques and tools for analyzing these properties of reliable distributed system designs. The recent approaches to system designing advocate that the design analysis activities should proceed concurrently with the design activity in a tightly coupled fashion; each design step needs to be validated to ensure that the design decisions would lead to the desired performance and reliability goals. This chapter presents a brief overview of the major accomplishments towards this goal. The presentation here is divided into three major sections. The first section describes the techniques for modeling fault-tolerant sytems using PAWS for the performance evaluation. The second section deals with the reliability analysis techniques, and the third section is devoted to the techniques for proving or validating the correctness of recovery mechanisms in distributed systems.

6.2 Performance Evaluation Of Recovery Mechanisms

The first concern of a system designer is generally the correct functionality of the system he is designing. It is, of course, imperative that a system correctly performs the tasks for which it is intended. Until very recently, designers did not concern themselves with the costs, in terms of resources and time, of providing this functionality until after some or all of the system was operational. Early performance predictions and the resulting design iterations are especially important in the design of highly reliable systems. This is because, unlike functional correctness, the reliability of certain functions or modules might be negotiable. If the cost of a reliable function is too high, the designer might be willing to accept a lower degree of reliability for that function which is not so extravagant with system resources. Such tradeoff decisions can only be made if the designer has at his disposal early estimates of performance and reliability.

The performance analysis part of the design evaluation phase is concerned with providing quantitative estimates for certain resource, utilization performance measures. Exactly which measures are interesting to the system designer and at what level of detail these estimates are to be made are questions for which it is important to have answers before proceeding with the analysis effort.

6.2.1 Performance Measures

There are several generic performance measures which are typically used to describe and quantify the performance of computer systems. These fall into two distinct categories: user- oriented measures and system-oriented measures.

The user-oriented measure most often used with respect to interactive systems is response time (turnaround time for batch systems). Response time is the elapsed time between the arrival of a request and the completion of that request by the system. Of course the exact moments of "arrival" and "completion" of a request must be carefully defined for any given application.

The two system-oriented measures most commonly encountered are throughput and utilization. Throughput is defined as the average number of requests processed by the system per unit of time. This is typically not a very useful measure of system performance since, as long as the system is performing well enough so that it can complete requests without creating an ever- increasing backlog, the throughput of the system is equivalent to the average arrival rate of the requests. Utilization is defined to be the fraction of time that a particular resource is busy - that is, working on some request.

6.2.2 Models and Hierarchical Structuring

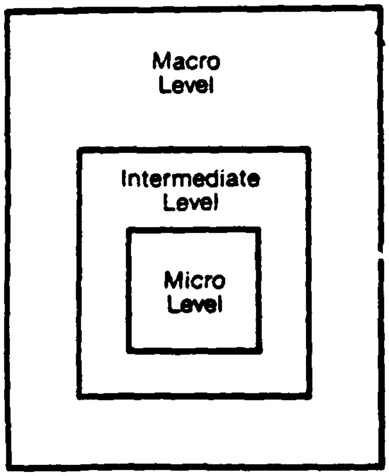
For operational systems, the most straightforward approach to performance evaluation is to directly measure the performance using some combination of hardware and software monitors. This, of course, is impossible during the design phases of a system since there is nothing yet to measure. In such cases when direct measurement is impractical or impossible, a model of the system must be devised which captures the salient factors that determine system performance. The model is then evaluated and the performance measures thus obtained are used as estimates for the performance measures of the actual system.

The complexity of such models and the degree to which they represent or abstract from the actual system determine to a large extent the amount of effort and expense required to evaluate them. Generally, the more detailed the model, the more expensive it is to evaluate. Luckily, during the design phases of a system, there is normally not a requirement for extremely accurate estimates of system performance. We are typically more interested in rejecting those designs which have a very negative impact on performance and in providing guidance as to which parts of the design should be considered for optimization; Therefore, performance models constructed during the design stage are normally simpler and more abstract relative to the actual system.

Even so, modeling a system which has many interconnected parts, even at a very abstract level, often produces overall models which are large, complex, and for which evaluation is intractable. The solution to this problem is the same as the classical solution to the general problem of software complexity: hierarchical structuring. Models which are decomposed into several smaller sections and structured vertically or hierarchically as in Figure 6-1 prove to be both more manageable and easier to evaluate [KOBA78] [BROW75]. Such hierarchical structuring allows the analyst to summarize the performance results obtained from evaluating one level of the model (say the micro level in Figure 6-1) in a form which is easily usable in the next higher level (the intermediate level in Figure 6-1).

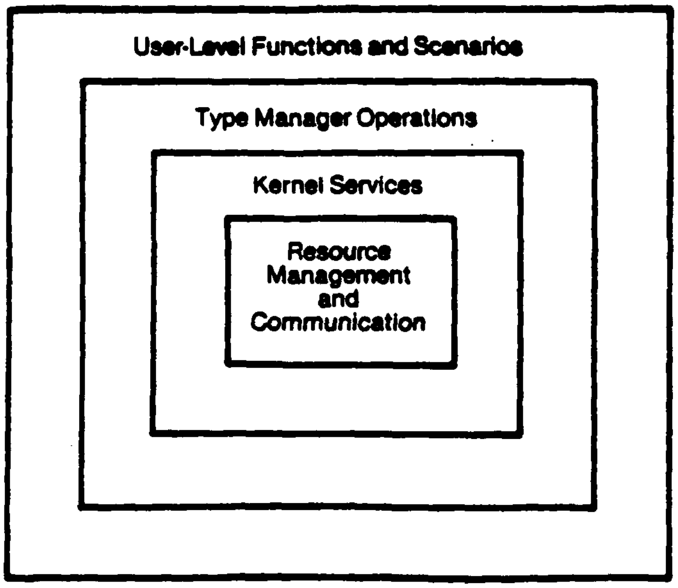
The decomposition of a model into a hierarchy of sub-models should take into account the inherent structures of the machine configuration as well as the system being modeled. A common rule of thumb criterion [KOBA78] is that the time constant at a given level of the model should be significantly smaller than the average inter-event times at the next higher level. In other words, a large number of state change events should occur at the lower level between events at the next higher level. In Figure 6-1, for example, the micro level models might have typical inter-event times on the order of micro-seconds or nano-seconds, the intermediate level on the order of milli-seconds, and the macro level on the order of seconds.

For object-oriented, high-integrity systems, there are a number of convenient levels of detail for which performance measures may be obtained. This natural hierarchy of modeling layers is illustrated in Figure 6-2.



Hierarchical Model Structure.

Figure 6-1



Object-Oriented Modeling Hierarchy.

Figure 6-2

6.2.3 Parts of a Performance Model: System, Environment, Workload

There are really three distinct factors which impact the development of performance models for computer systems. The most obvious of these is, of course, the structure of the system which is to be modeled. As previously mentioned, the structure of the model might not operationally reflect the structure of the actual system but might rather abstract from it the main features which affect performance. It is believed, however, that the object-oriented system structure discussed widely in this guidebook will simplify the task of producing performance models of the system. This is because of some of the same reasons that make this approach highly suitable for the formulation of highly reliable distributed systems: inherent modularity and hierarchical structuring.

In addition to modeling the structure of the system, the environment in which the system must operate must also be considered. The environment includes such things as the native hardware and software in which the system is to be embedded and, of particular interest in the modeling of reliable systems, the fault characteristics of that hardware/software configuration.

Finally, the workload which the system will be expected to accommodate must also be modeled in some way. Choosing an appropriate workload and a representation for it is less of a problem for existing operational systems although it is still very much an art and still very difficult to do. For systems which are not yet operational, the problem becomes one of choosing or inventing a hypothetical workload which will hopefully reflect the characteristics of the future workload of the actual system.

In order to obtain useful predictions of performance measures from the models, they must be evaluated in some way. Performance model evaluation is the process by which values are derived for the chosen performance indices given a "correct" and properly parameterized model and an appropriate workload. Once a validated model has been constructed, it must be evaluated to obtain values for the performance indices of interest. Models may be evaluated analytically, by simulation techniques, or by some hybrid combination of the two.

6.2.3.1 Analytic Methods

In [KOBA78], an analytic evaluation method is defined as, "a solution technique that allows us to write a functional relation between system parameters and a chosen performance criterion in terms of equations that are analytically solvable." The term "analytically solvable" here is usually taken to include numerical solution methods other than simulation as well as closed-form solutions. Although such a definition of analytic solvability includes deterministic techiques like automata theory and Petri nets, the term is most often used to refer to the mathematical discipline called queueing theory. Mathematical queueing theory provides a framework in which networks of resources (CPU, memory, I/O devices, etc.) are being prevailed upon by jobs to perform some services. Contention for a resource causes jobs to be queued for later service.

6.2.3.2 Simulation Methods

When evaluating an hierarchically structured model of a large system, it is likely that at least some of the submodels will be susceptible to analytical methods. It is also very likely, however, that the analytical solution of some of the submodels will remain mathematically intractable even with simplifying assumptions and constraints. In these cases, the only alternative evaluation method for non-existing systems is simulation.

Simulation is a numerical technique for evaluating queueing network models by mimicking the dynamic behavior of the system being modeled. The principle advantage of this technique is its great generality. Most of the constraints which are necessary for analytical methods have little consequence with respect to simulation. The three main problems with simulation are the expense involved with building the simulator, the expense of running the simulation, and the necessity for statistical analysis of the resulting output data.

The problem of the expense of running simulations derives from the fact that the length of a simulation run is proportional to the number of events which must be simulated rather than to the duration of simulated time. In the example of Figure 6-1, it would probably be desirable to run a simulation of such a system long enough to see perhaps hundreds of events at the macro-level in order to ensure that the simulation reaches a steady state. Since events at this level occur approximately every second, we will wish to run the simulation for something on the order of say 1000 seconds. But if the intermediate and micro levels are also entirely included in the model, the total number of micro events which must be simulated might be on the order of several billion. Such a simulation run will likely be very expensive. This problem is most effectively controlled by hierarchical structuring. This allows low-level models to be evaluated separately and the results summarized at the next higher level in the form of a scaling factor or statistical distribution.

6.2.3.3 Hybrid Methods

A combination of both analytical and simulation methods may be used in evaluating a model of a large system. Again, the hierarchical nature of the model may be taken advantage of to allow lower-level sub-models to be evaluated using either analysis or simulation whichever is more appropriate and least expensive. The results thus obtained may then be summarized in modeling the higher layers.

6.2.4 Performance Measures for Recovery Mechanisms

The design tradeoff decisions concerning reliability and integrity mechanisms and performance are generally more complex than those for conventional systems where high reliability is of less importance. Such tradeoffs are conventionally between different kinds of performance, such as resource utilization and response time. The only other analytical property of such systems is their correctness - the degree to which they satisfy their operational specifications. For obvious reasons, the correctness of a program is rarely purposely compromised in favor of better performance. It may, however, be perfectly valid to design an object so that it is slightly less reliable but responds quicker (or vice versa).

Because of the complex tradeoff decisions which are likely to be involved in configuring a system such as the one with which we are currently concerned, it will be necessary for the designers (and possibly also the system administrators) to have at their disposal reasonably accurate estimates of the costs of the various reliability and integrity mechanisms which are provided by the system. In order to provide such estimates, the analyst/designer must examine at least three different cases:

- The performance of the system in the absence of the relevant reliability/integrity mechanisms.

- The performance of the system with the relevant mechanisms in place but in the absence of the failures which the mechanisms are there to protect against. This class of performance figures, when compared to those obtained as above, will provide a useful estimate of the best case cost of providing protection from faults.

- The performance of the system with integrity mechanisms in place and when failures of the defined class actually occur. Together with the results obtained in the first two cases above, these figures will provide an estimate of the time and resource requirements of the recovery mechanisms of the system.

6.2.5 Example Metrics for Some Generic Integrity Mechanisms

The following is a sampling of some of the performance measures which are likely to be interesting in a distributed, object-oriented reliable system. Three generic classes of reliability and integrity mechanisms are used to illustrate the issues involved; transactions, concurrency control, and object replication. A more detailed discussion of these mechanisms may be found in Chapter 4.

6.2.5.1 Transaction Mechanisms

Of course, user-level scenarios will probably be defined as or in terms of atomic transactions. The response times and throughput of these will be of primary concern. In this section, however, we will be dealing only with the low level performance characteristics of the mechanisms used to attain atomicity and reliability for groups of associated individual type manager operations. The following is a list of some of these low level characteristics:

- Mean Rollback Time - The mean time required for a type manager to rollback an object to a previous state (the state of the object at the time of the last checkpoint).

- Mean Size of "Window of Vulnerability" - The mean time during which an object is vulnerable to a failure of the coordinator of the transaction.

o Mean User In-Doubt Period - The mean time from when a user decides to commit a transaction until the user can be told that the results are committed.

o Mean Coordinator In-Doubt Period - The mean time (from when a coordinator issues a commit message until it receives acknowledgments from object managers, e.g., second phase) during which a coordinator must retain the state of a transaction.

6.2.5.2 Object Replication

There are both costs and benefits associated with maintaining multiple redundant copies of some objects. The costs are in the form of the additional storage requirements for the redundant copies and the time and resources required to ensure that the multiple copies remain consistent. The performance benefit stems from the fact that, in some cases, local copies of an object may be used to provide read-only access thus eliminating the communication costs of accessing a remote copy instead.

o Redundant Storage Overhead - The additional storage and other resources required to maintain all but one of the identical copies of an object.

o Multiple Update Overnead - The additional time and resources required to update additional copies of a replicated object.

o Read-Only Access Improvement - The average improvement in read-only type operations due to the distribution and replication of an object.

6.2.6 Zeus Performance Modeling

The performance evaluation of Zeus is carried out using PAWS (Performance Analyst's Workbench System) [IRA83], a general purpose simulation language for the performance evaluation of system models. Our choice of these particular tools was partly due to in-house familiarity with them (PAWS is a registered trademark of Information Research Associates), and partly due to their suitability for the tasks of representing and evaluating performance models.

The creation of a performance model is a multi-step process involving first a determination of the most relevant execution pathways in the system design (i.e., there are many possible execution pathways in any system, and only a subset of those is used with great frequency). This shifts the focus upon those modules and the system activity those modules represent that is most relevant to the performance of the system. Once the performance determining pathways are defined, they must be coupled in a meaningful fashion with a target resource (hardware) configuration and a specification of the resource usages along the performance pathways.

Execution paths are translated into In^-rmation Processing Graphs (IPGs), which are pictorial constructs for modelin information processing systems. As given in an introduction to this tool [IRA83], IPGs are a useful modeling methodology for several reasons: pictures often provide the best method for describing and understanding information flow; it is easier to communicate ideas quickly using a picture; and information processing systems are often designed around a structure of information flow. From these IPGs, it is a straightforward translation to a queueing network model.

In a distributed operating system, information flows through resources on hosts and between hosts in the network. The basic graphical components are nodes, edges, and labels. In an IPG, each node represents a resource (such as CPU, memory, disk units, etc.) while edges connect nodes and represent some form of information flow from one resource to another along an edge. Edges are given labels denoting the form of information flow. The IPGs are directly mappable to the Performance Analyst's Workbench System (PAWS), which is a simulation language that is used to evaluate performance models. In a model, the information flows which are of interest are given what are termed category and transaction names for which statistics are gathered during the simulation. Additionally, for each resource in the model a set of summary statistics is generated.

6.2.7 Summary

In the system designers guidebook we have discussed the tools and techniques that are available to aid in the performance analysis of distributed, reliable systems. We began by very briefly surveying the field of performance analysis of computer systems, especially emphasizing the issues that were relevant to performance analysis during the design phases. Modeling and model evaluation techiques are the important topics in this regard. In addition to the general sketch of performance modeling, we also give a somewhat more detailed account of several of the representational and simulation tools. The specific issues involved with modeling the design of a particular class of computer systems are discussed in the guidebook. It should be apparent from the material in this chapter that performance evaluation during the early stages of design and continuing throughout the lifetime of the system can be an invaluable strategy for producing viable, efficient software products.

6.3 Reliability Analysis Techniques

Similar to performance characteristics, the specification and evaluation of reliability characteristics of a design are an important and integral part of the design process for reliable systems. A design process typically consists of several phases starting with the requirements specifications up to the final design meeting those requirements. These phases may involve several iterations of designing and validation until the design meets the desired requirements. For reliable systems, the requirements statements must include the specifications of the desired reliability characteristics of the target system. Typically a design process consists of decomposing the design into a set of sub-problems. In such cases the requirements statements, which include the reliability specifications, are appropriately extended and augmented for each of the sub-systems. The validation task consists of verifying that the target system constructed from those sub-systems, with the given reliability characteristics, has the desired reliability.

The traditional approach to specifying reliability characteristics is to use certain numerical measures such as availability, mission time, and mean-time-to-failure (MTTF). Chapter 2 described some discrete measures for reliability specifications. These measures imply that a system has certain failure characteristics under a given set of system faults. The measures capture the level of consistency maintained by the system under this set of faults.

There are essentially two approaches to reliability analysis of system designs -- simulation and analysis. One approach is to simulate the system design along with its failure environment and the recovery mechanisms. This approach is inherently expensive because it requires building simulation models specific to the design to be analyzed. This approach provides relatively more accurate results as compared to the second approach because it captures the structure and functioning of the system to a greater detail. The second approach is based on combinatorial analysis of the system based on the reliability characteristics of its components and their interconnections. The reliability characteristics of the components are specified in terms of availability and MTTF. This approach is, in general, faster and less expensive.

The combinatorial analysis methods provides quick first-order evaluations of the system reliability characteristics given the system configuration and the reliability characteristics of its components. These methods can be used to construct a general purpose evaluation tool. One such tool called NetRAT (Network Reliability Analysis Tool) is described in this chapter. The evaluations using this method are somewhat less accurate as compared to using simulation models bacause they do not capture some of the dynamic operating conditions such as execution delays, system load, and resource contention.

6.3.1 Specifications of Reliability Measures

Traditionally the reliability characteristics of a system are expressed in terms of certain probabilistic measures such as the availability, reliability, mean-time-to-failure (MTTF), and mean-time-to-repair (MTTR) for repairable systems, mission time, etc.

For a large system, such as a distributed command and control system, rather than specifying the availability and MTTF of the entire system one would be more realistic in individually specifying the reliability characteristics of its services and virtualized resources as seen by the system users. The approach that we follow consists of specifying the reliability characteristics of the functions executed on the system objects. These characteristics for a function will be different for its invocations from different nodes. For example a system service might be available 100% of the time when accessed from one node, whereas the same service might be available for only 90% of the time when accessed from some other node.

The availability A(t) of a system is a function of time indicating the probability that the system is functioning correctly at any given time t. In distributed systems, we are interested in computing the availability of the functions (services), which expresses the probability of that function (service) being available at any random instant of time. A function execution at a node requires access to some resources which are distributed in the network. A successful execution of the function requires that the resources be accessible from the node where the function is being executed. Therefore, the availability of a function is dependent on the availability of (1) the communication paths to the required resources, (2) the nodes holding the resources, and (3) the nodes executing the function.

The mean-time-to-failure (MTTF) for a service in a distributed system is the expected time interval during which that service remains available before a failure occurs. A service fails if it is unable to access any of the required resources or if the node executing the service fails. MTTF is an important measure of reliability in distributed systems because of the possibility of large delays encountered in communication.

Using the numerical reliability measures for requirements specifications raises certain problems. One of the problems is dealing with small numbers in specifying these measures. Another problem is related to the fact that the reliability measures of the system components are significantly altered in the combat conditions. Under such circumstances the reliability analysis techniques should focus on determining whether the system performs correctly and in a timely fashion if a certain set of resources are unavailable. This leads to specifying a discrete set of reliability levels corresponding to the consistency levels maintained by the system under various fault conditions within the system. The system designers guidebook presents four discrete reliability classes for objects.

6.3.2 Network-Based Reliability Model

This section describes a network-based approach for representing a system to evaluate its reliability. This model ideally suits for representing distributed system architectures. In the past a considerable amount of work has been done in the evaluation of reliability and availability of paths in network-based systems, particularly in the area of communication networks. Most of this work addresses the problem of pair-wise terminal reliability in communication networks. i.e., given a pair of nodes in the system, determine the availability of the communication path between these two nodes.

In distributed systems, an important generalization of the pair-wise terminal reliability problem considers the availability of paths from a set of nodes in the network to a different set of nodes. For example, a service execution in a network might require access to several resources that are located at different nodes. It is also possible for a service to require access to any one of the several resources distributed in the network. For example, a read operation on a replicated file can be successfully performed if the node executing this operation can reach any one copy of the file. This is referred to as the multi-terminal reliability problem and has been addressed in a recent work [GRNA81]. In [GRNA81] an algorithm is presented which computes the multi-terminal availability from the availability of the network components.

ANALYSIS AND VALIDATION TECHNIQUES

NetRAT is a reliability analysis tool for network-based systems, which facilitates the evaluation of multi-terminal reliability characteristics. The NetRAT system is essentially based on the algorithms described in [GRNA81] [GRNA80]. However, the algorithm presented in [GRNA80] is incorrect. We have corrected this algorithm in [WANG83] and incorporated it into NetRAT. In addition to the availability calculation, NetRAT also permits the evaluation of other reliability measures, such as the reliability function, mean-time-to-failure (MTTF), and mission time. These extensions are described in the next section.

The reliability analysis model underlying the NetRAT system is network-based, and the evaluation procedures are combinatorial. In the network-based model, a system is represented as an interconnection of nodes. The nodes represent the functional units; and the links, which can be either directional or bidirectional, represent the communication paths. Reliability measures such as availability, reliability, MTTF, etc., are associated with these components. In the NetRAT model, a set of functions and resources are assigned to these nodes. Each function requires access to some resources, which can be physical resources or other functions. Functions in the NetRAT model correspond to activities which provide services in real systems; and physical resources in the NetRAT model correspond to data and hardware resources in real systems, such as processors, memory, disks, I/O devices, files, etc.

A node may contain more than one resource or service, and multiple copies of a resource may exist at several different nodes. In case of multiple copies of a resource, any one of these copies can be used to meet the resource requirements of a function. A function may be available at several different nodes. The resource requirements of a function can be combinatorial; for example, a function may require resources (A and B and C) or (B and D).

We illustrate this network-based model using a set of examples. Consider the network shown in Figure 6-3. This network model consists of four nodes 1,2,3, and 4. The availability data of these nodes and the interconnecting links are shown in the figure. A function (program) called FUN executes at node 1. This function requires access to resource R1 and R3. Resource R1 is located at two nodes, 2 and 4, and resource R3 is located at nodes 3 and 4. In this example, we are interested in computing the availability of function FUN. In the model shown in Figure 6-3, if a node is available (functioning correctly), then all the resources located at that node are available. Consider another scenario in which a node may be available, but the resources located at it may not all be available. At a given node, the availability of a local resource could be less than 1.0. For example, in the system of Figure 6-3, resource R2 at node 3 is available with probability 0.7 and R3 with probability 0.8. In order to represent this system in the NetRAT model, the network model in Figure 6-3 is changed to that in Figure 6-4. Here resources R2 and R3 are represented as separate nodes (shown as nodes 5 and 6) connected to node 3.

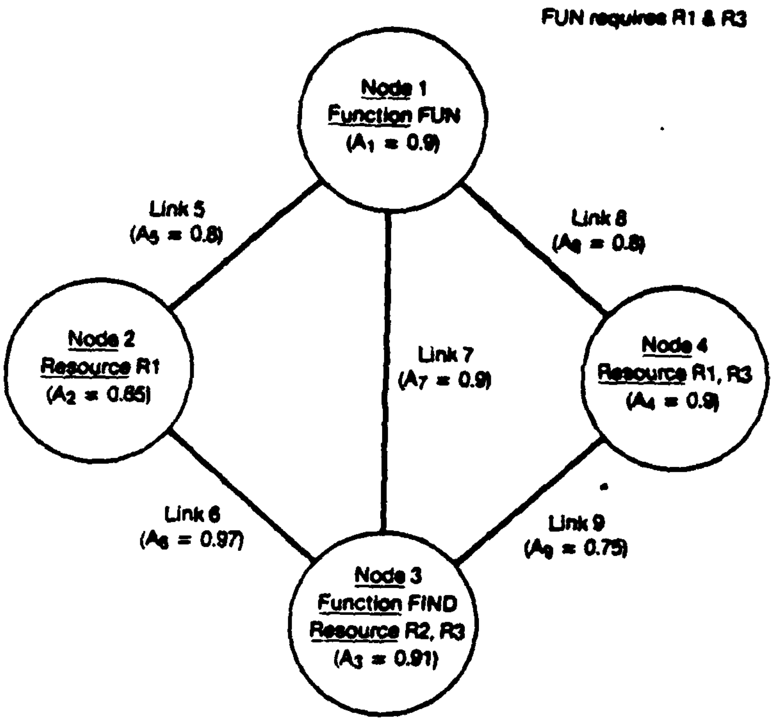


Figure 6-3

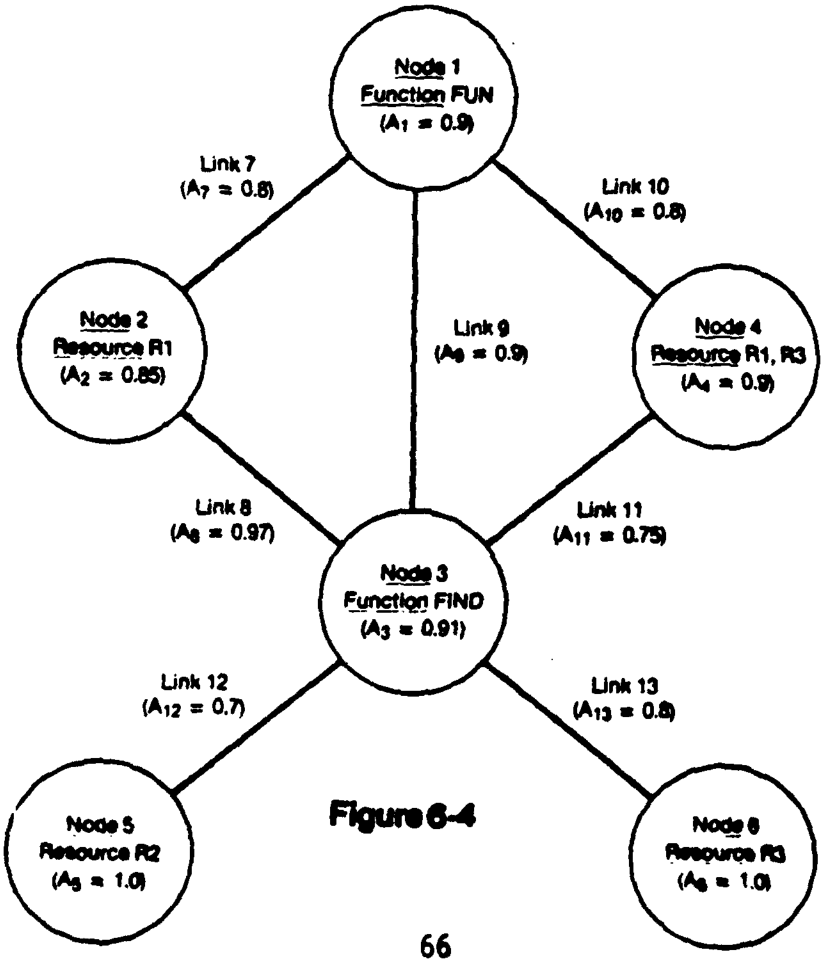


Figure 6-4

As mentioned earlier, it is possible to include in the resource requirements of the function FUN some other function names. The actual physical resources required for FUN include the union of the resources required by each of the fuctions whose names appear in the resource requirement of FUN. For example, in a modified scenario the function FUN requires resources R1, R2 and FIND, where FIND is a function that requires the resource R3. Hence the total resource requirement for FUN consists of R1, R2 and R3. Recursive references to function names in the resource requirements are permitted, as long as they can be resolved in terms of physical resource requirements.

6.3.3 CONCLUSIONS

In the system designers guidebook, we have presented a modeling and analysis method to evaluate the reliability characteristics of systems. The analysis is combinatorial and it is not easy to use manually. It is advised that such procedures be automated as a general purpose tool. A system called NetRAT, which is based on such a procedure, has been described in the guidebook. The modeling approach used in NetRAT is network based, i.e., the system is viewed as a collection of nodes connected by either bidirectional or unidirectional links. Reliability characteristics of the individual links and nodes are used to determine the reliability characteristics of the composite system and is particularly attractive from the viewpoint of hierarchical analysis of systems.

6.4 Validation And Verification Techniques

In this section we describe three methods for proving or analyzing the fault-tolerant properties of distributed system designs. The detailed descriptions of these methods can be found in the first volume of the system designers guidebook. The first method is based on applying program verification techniques to the design expressed in a suitable programming language. This method is, in general, expensive in terms of time and it necessarily requires support of automated tools during the verification process. This involves proving certain formally stated properties of the software system. During the last decade a considerable amount of work has been done in the area of developing languages and their support tools that facilitate formal verification of the software. The most notable of these systems are Affirm [GERH80], HDM [ROBI75], FDM [KEMM80], and Gypsy [GOOD78]. The Gypsy language and its verification system have been designed to facilitate verification of communicating processes. This makes Gypsy an attractive candidate in this category of methods. In this section we present a brief description of the application of the Gypsy methodology to proving the fault-tolerant characteristics of a system that is structured according to the design model presented in Chapter 4. Application of this technique is suitable when the design has been refined and specified to a detailed level. The examples presented in Chapter 10 of the guidebook deal with the recovery mechanisms at a single site.

The second method for proving fault-tolerance of distributed system designs is relatively less rigorous and is amenable to manual proofs for small systems. Nevertheless, this method can be developed into a computer-assisted system. In this approach we focus on proving properties of a set of communicating processes. The system is abstracted as a collection of finite state machines which interact by exchanging messages. The proofs are based on the properties of the state sequences of these machines and the relationship among the state sequences based on the communication events. Each finite state machine is specified in terms of events and state transitions. Detailed descriptions of the system in a programming language are not required at this level; therefore, this method looks attractive at the higher-level design phases such as the conceptual design or the functional architecture design. An example dealing with the proof of a two-phase commit protocol is presented in the guidebook.

The last method of design validation for fault-tolerance is based on functional simulation of the design. In this method, to validate certain recovery characteristics of a system, simulation models of the appropriate parts of the design are constructed in a suitable simulation language such as Path Pascal, Simula or PAWS. The simulation models mimic the functional behavior of the actual system as intended by the design. Some of the basic issues involved in simulating the fault-tolerance characteristics of a design, the requirements on the simulation language for this purpose, and the salient features of a Path Pascal simulator for the Process/Transaction Manager in the Zeus system. This approach is expensive in terms of time and effort because it requires building exact simulation models of the system components.

6.4.1 Proofs of Recovery Mechanisms using Gypsy

6.4.1.1 Introduction

Gypsy is a mature methodology for constructing formal proofs that a software system satisfies formal specifications [GOOD78, GOOD82a, GOOD82b]. Gypsy has been applied very successfully in several security applications, but no attempt has been made to apply Gypsy to recovery problems. The focus in this effort has been to answer the question, "What can be specified and proved about recovery mechanisms with the existing Gypsy methodology?"

The designers guidebook describes two examples based upon work described in Chapter 4. The first set of three examples illustrate different Gypsy implementations of recoverable objects. A generic shell formally specifying the behavior of recoverable objects is given, and then three different implementations are shown to satisfy the formal specifications of the shell. The second example is a Gypsy model of a transaction recovery scenario given in Chapter 4. The recovery scenario is modeled in Gypsy, and formally specified. This example supports the position that the precision required to write formal specifications has the potential to contribute significantly to the quality of the resulting design.

The intent of the recovery mechanisms considered in the effort related to formal verification of recovery mechanisms is to provide necessary support the implementation of atomic transactions. An atomic transaction is an operation with the property that if it fails, the data objects that it was altering are restored to the values that they had when the transaction first accessed them. In a distributed system there are the added complications of multiple copies of data objects which must be maintained in synch, transactions which may be spread over multiple machines, and host crashes and message transmission errors.

The process of developing formally specified code is not radically different from the standard cycle of software development. The critical difference lies in the use of formal specifications. Because they are so precise, formal specifications are often more difficult to write than an English statement of a functional specification. However due to this enforced precision the resulting specifications are considerably more useful. Additionally, the requirement that the code be proven to coincide with the specifications provides a tremendous increase in confidence in the resulting software.

One point must be emphasized. Formal specification and proof does not guarantee that there will be no errors in the code. It is possible that the specifications do not capture the designer's intentions. It is also possible to specify only part of the functionality of a program, in which case the unspecified portions of the program may go wrong. What the verification process does assure us is that the specification and the code are consistent with each other.

6.4.1.2 Gypsy Support for the Specification of Recovery Mechansims

Gypsy provides a number of mechanisms that support the verification of recovery mechanisms. There are two basic sorts of approaches that can be taken, which are reflected in the two examples in this section. One is an object oriented approach, which makes use of Gyspy's standard specification methods, in this case lemmas to algebraically specify object properties and routine specifications to specify the effects of operations on objects. Gypsy's abstract data type facility could also be used effectively for these examples . this approach one can describe the required properties of the selected objects and then demonstrate that the proper selection of procedures to manipulate these objects maintains this set of specified properties.

The other approach is to develop a procedural model that takes advantage of Gypsy's concurrency mechanisms to simulate the distributed world, with buffer operations to carry message traffic. Buffer histories are used to specify such systems.

These two methods are complementary. On the one hand the object oriented specifications provide a mechanism to specify the properties that recoverable objects must have. A procedural model then permits the verification that the procedures designed to maintain these objects in a proper state function as intended.

6.4.1.3 Specifications and Proofs of Recoverable Objects

The example presented in the guidebook is chosen from the design model for reliable distributed systems presented in Chapter 4. We have chosen the recoverable object level of the model, which is built upon stable objects, and supports atomic transactions. In this example the stable objects are of arbitrary type (left "pending" in the Gypsy notation).

We do not concern ourselves with the issues of security and access control identified in the model. The problems of transaction and process management would be dealt with outside of the portions of the system modelled here. We also do not consider the problems of implementing stable objects, but construct recoverable objects out of a data type, which is an abstraction that is left pending.

First, formal specifications of recoverable objects are given. Then three different implementations of recoverable objects are presented, and proven to meet the required specifications. Finally, we give two examples of how this model might be extended to cover two more abstraction layers of Figure 4-2. The effects of incorporating the stable object layer beneath the recoverable object layer on the proofs are discussed in detail in the guidebook. The description of recoverable object can be used as the basis for the next higher level handling of atomic transactions. The example builds a small type manager that employs a simple locking protocol based on the recoverable object specification.

6.4.1.4 Recovery Scenario for Atomic Actions

The study of applying Gypsy methodology to proving atomicity of transactions in the example system demonstrates the utility of modeling system designs in Gypsy. Even in the absence of proof, the need to write specifications precise enough to support critical inspection forces a detailed examination of assumptions. While this is a subjective process, as opposed to the objective nature of the proof process, it increases the likelihood that the coverage of the specifications is sufficient to describe the behavior of the system under all cases included in the top level specification. In other words, the specifications on the various components of the system are likely to support our expectations (as embodied in a top level specification) about the system as a whole.

6.4.1.5 Summary

Based on this experience, we offer the following observations.

1. Various aspects of distributed command and control systems can reasonably be described (formally specified and implemented) in Gypsy, and the implementation verified against the formal specification. The Gypsy implementations may well serve only as models for actual implementations in other languages, but such efforts should significantly increase confidence in the correctness of the resulting code.

2. Some elements of these systems do not map directly into Gypsy. For example, the notion of spawning a process is not supported by the Gypsy model of process invocation. Thus, some pieces of the system design can only be modelled in Gypsy in a fashion quite different from the intended implementation. We believe, based on our own experience, that composing these models, and formally specifying and verifying them can have considerable benefit in enhancing the designer's understanding of the

system, and the level of precision in the system description. Even in the absence of formal proof, the additional precision supplied by formal specification can be of utility.

6.4.2 Recovery Mechanism Proofs using Interval logic

Verification techniques based on analysis of input and output message streams [MISRA81] and message buffers [GOOD79] suffice for establishing "black-box" stimulus-response behavior of a process or network of communicating processes. However, an important class of properties -- relationships among system state variables -- cannot be as easily expressed and verified with these methods. We would like to be able to combine assertions over the states of several processes, so-called "local" assertions, into a system-wide "global" assertion stating a relationship among the variables of the several processes. The difficulty is that such an assertion is intended to hold at some particular "time", i. e., point in the history of the system, and this requires a rough synchronization or "lining-up" in time of the various processes. If the assertions are construed as holding at some instant of time then it is required that the processes be precisely synchronized so that at that instant all the related variables are stable and in the desired relationship. This precise synchronization is difficult to verify and can be expensive for the system to arrange.

The primary contribution of this work is the development of a framework that facilitates construction of global assertions from local assertions. The following section presents in an informal fashion the approach for constructing global assertions about the communicating processes in a distributed systems from the local assertions of individual processes. This approach examines behavior of such processes over certain intervals, establishes relationship among the intervals, and then derives global assertions using these relationships. Because the process behavior is described over intervals, we find use of temporal logic notation [MOSZ83, HALP83] convenient in such proofs.

6.4.2.1 Proofs of Global Assertions

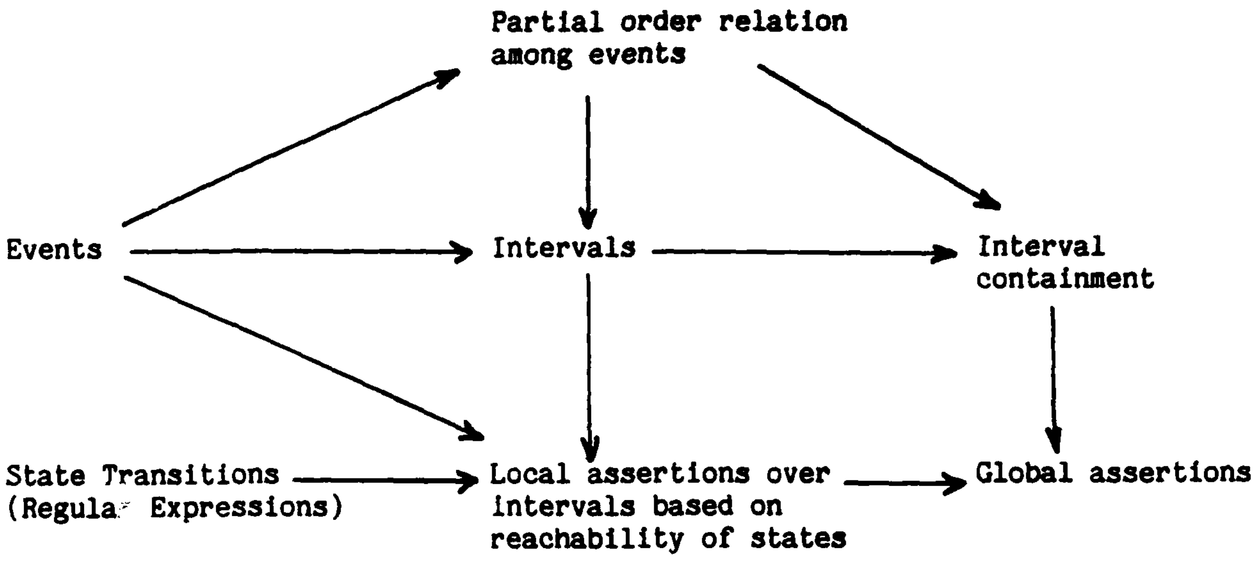
The approach presented here is intuitively quite simple. In this method each communicating process is viewed as a finite state machine. The state transitions in such a finite state machine occur either due to some internal or external events. The external events correspond to the arrival of a message from some other process. A process in a given state maintains certain assertions over its variables. We refer to such assertions as the local assertions. The occurrence of some event may cause a process to enter a new state; during this state transition, the process may execute certain actions which lead to new events in the system. Some of these actions - those which send messages to other processes - may cause occurrence of events in some remote processes.

A sequence of state transitions in a process can be represented as a sequence of states. Such a state sequence also represents the behavior of the process over some interval in that process's life-time. If a local assertion holds true in each state of a state sequence, then we say that that assertion holds over the state sequence. This leads to another way of characterizing intervals in a process's life-time as the state sequences over which certain assertions are maintained. Therefore in the rest of this report we use the term interval to characterize those state sequences which maintain some assertion. In case of finite state machines, state sequences of interest can be identified by constructing the regular expressions for the machine. The interested readers should look into some text-books on automata theory for this purpose. These regular expressions also define the reachability sets for the states.

An important step in deriving global assertions on the basis of the local assertions of individual processes over their state sequences is to establish relationships among those state sequences (or, intervals). These relationships define if an interval precedes or is contained in some other interval. Such relationships are established on the basis of the communication events among the processes. Using a partial ordering model for events in distributed system, such as the one presented in [GREI78], one can establish precedence and containment relationship among the local intervals (state sequences) of the various processes in a distributed system. An interval I1 is contained in another interval I2 if, and only if, the first event of I1 precedes the first event of I2, and the last event of I2 precedes the last event of I1.

A global assertion in a distributed system relates local variables of several processes over some intervals in the life-time of that system. The important step is the conjunction of several local assertions over the same interval. Suppose that an assertion p holds true for the local variables of some process P during some local interval I1, and an assertion q holds true for the local variables of some process Q during some local interval I2. Now suppose that I2 is contained in I1. This means that during the interval I2 (which can be viwed as a global interval for P) the assertion p is true for process P. Therefore, the assertion (p and q) is true for the set of processes P and Q during the interval I2. The validity of this statement is quite obvious.

The method for proving global assertions in this approach is schematically shown in Figure 6-1. The partial order relation between events defines intervals and the containment relationship between intervals. A local assertion for a process during an interval is derived from the set of reachable states during this interval. The reachability set during an interval is computed from the initial state during the interval and the state transition specifications along with the events that can possibly occur during this interval.


Partial order relation among events
Events
Intervals
Interval containment
State Transitions (Regula- Expressions)
Local assertions over intervals based on reachability of states
Global assertions

A Schematic Representation of the Proof Method Using Intervals

Figure 6-5

The first step in the method is to specify each process as a finite state machine. This requires definition of the states and the state transitions under various events. The set of events also include some communication events, such as sending or receiving a message. For each process, based on its finite state machine description, the regular expressions are constructed. There regular expressions are used for reachability analysis during the proofs. The next step involves identification of the intervals of interest for which certain properties are to be proved; this requires a clear understanding of the problem. These intervals are in general subsequences of the regular expressions for the finite state machine. Relating the intervals of different processes for global reasoning is done on basis of the communication events.

6.4.3 Functional Simulation of Fault Tolerance

Functional simulation is an approach for validating that a model of a software system exhibits a desired property. In this section we discuss the use of functional simulation techniques for validating that a software system is fault tolerant. The discussion is based on our experience building a Path Pascal model that simulates a subset of the Zeus process/transaction manager and a subset of a generic object manager. The fault tolerance property that is validated is that transactions do provide an "all or nothing" effect even if site crashes occur and messages are lost or duplicated.

Functional simulation is one of the few techniques that permits the early examination of the behavior of a program. The costs associated with this technique are directly related to where in the lifecycle the activity is performed. The closer to implementation, the more detailed a model will be, more costly to develop, and difficult to validate and analyze, but the greater the potential insights and benefits. Functional simulation is a form of testing and does have the same disadvantages of testing. It can show the presence of anomalous behavior but cannot prove the absence of anomalous behavior. A survey of different approaches to software verification and validation and their strengths and weaknesses is given in [ADRI82].

Functional simulation uses an executable model to represent the behavior of an object for the purpose of analyzing whether the object correctly exhibits a desired property. Validation consists of observating the behavior that a model of an object exhibits when executed on models of a computational environment and external environment and analyzing that behavior with respect to the desired behavior of the software system. For example, if the property is security, the object modeled may be an operating system kernel. The validation may consist of observing which requests are granted and denied access; this can then be compared with what a model of the security property defines as correct behavior.

6.4.3.1 Issues in Simulating Fault Tolerance

Simulating fault tolerance and distributed systems raises a number of issues which impose requirements on the simulation system selected. This section discusses some of the technical difficulties that we have encountered and their implication for different simulation systems. A discussion of solutions to the problems presented in this section and of the actual model are given in the system designers guidebook.

The technical difficulties that arise are directly related to the property that is to be validated and the technique that is to be used for validation. Clearly validating the security of a design will involve different modeling issues and validation techniques. The following discussion is restricted to fault tolerance and specifically to validating the atomicity of transactions, although some of the discussion is relevant to the modeling and validation of properties in general.

The key events to be modeled are failures, so it makes sense to examine what their impact is on a model. There are a number of failures that are of interest. Among the kinds of failures are site crashes, memory failures (both primary and secondary), link failures, and lost and duplicate messages. The requirements that failures impose on a model may be analyzed by examining their effect on a computation.

A site crash results in all active computations halting at the same time in an unknown state. A model must be able to represent multiple concurrent activities and control their progress. The multiple concurrent activities model a system executing a certain number of user processes (e.g., equal to the desired multiprogramming level) and system processes. Controlling the computation's progress includes stopping a process when an event occurs and continuing the process when some other event occurs. There are two types of events of interest -- resource coordination and processor failures. Resource coordination may occur if multiple processes are sharing a resource (such as a processor or a file) or are cooperating to complete a computation (such as a buffer for a producer and consumer). Event coordination may be achieved by a synchronization object and mechanisms (e.g., a semaphore and the ability to allocate it and block processes). Processor failures require the ability to stop all processes simultaneously and to cause them to make a transition to a well defined next state.

It is not acceptable to put the burden of event management in the application and system processes. A process should not check the state of a resource each time that it accesses it, and it should not check to see if the processor is active before it executes an instruction. Ideally, an application process should request a resource and any synchronization should happen as a side effect. Similarly, if a site crash occurs all processes should be halted as a side effect of the failure and not due to the processes checki·g the processor state.

Simply halting the progress of a computation is insufficient for simulating a site failure. The state of a computation is divided between secondary and primary storage. Almost all primary storage is volatile. Hence when a site crash occurs, a certain amount of the state of a computation is lost. This requires that a model of volatile primary storage exhibit the loss of information. However, if a model simulates memory failures it still needs to be able to simulate recovery which requires starting a process at a defined point with its computation in a state that is consistent with that point. It seems as though a model must take snapshots (e.g., checkpoints) of a computation's evolving state and correlate those wi'. different points in the computation's progress. Two problems arise: there may be an arbitrary number of such points and how a modeler knows which ones to select.

Memory failures may occur independently of site crashes. This results in the same problems as above but with the added difficulty of only part of the state of a computation being lost (e.g., the state may be resident in multiple primary memories).

There are three kinds of communication failures of interest -- lost messages, duplicate messages, and link failures. All of the failures change the effects of operations on objects of type message. The first two occur intermittently and affect a single message. The last one occurs for a time interval that encompasses many messages.

A model of a failure should include the ability to do fault de ection and should not consume measureable resources or affect events that are independent of it. Link failures and memory failures are ex mples of resources becoming u navailable for a period of time. During that period the failures should be detectable, for example by timeouts or parity checks. A model of a site failure should not consume resources, so any computation done to simulate the failure cannot consume memory or CPU resources. Similarly the discarding of messages to simulate the loss of a message should not impact the CPU utilization measured.

Faults may be injected into a simulation either deterministically or probabilistically. The deterministic approach requires an explicit statement as to when a fault is introduced. It may be signaled either by an explicit call or parameter within a call. It may be triggered based on the state of the system. For example, if 10 transactions have been successfully completed, lose the prepare message issued by the transaction commit coordinator; or if there is an object in the commit pending state, then crash the site.

The probabilistic approach injects faults based on a distribution that is independent of the current state of individual computations. The signaling of probabilistic fault injection is done implicitly by a routine generating a value based on a distribution and determining whether or not the value generated implies that a fault should be injected. For example, a network driver routine may generate a value based on a uniform distribution. If the value falls within a specified range, a message is lost.

The selection of which approach to fault injection to use is made based on what is to be learned from a model and how much effort is to be spent building and analyzing a model. The two approaches must be matched to the use of the model and the kinds of faults that must be injected. For example, if a model is to be used to demonstrate that a communication subsystem provides reliable message delivery, faults in the form of lost and duplicate messages may be introduced probabilistically. Other times it may be more important to see the impact of a specific set of events on an operation, for example, the effect of losing a specific message, such as a commit message, on the timeout period and window of vulnerability for a commit protocol. This example requires deterministic fault injection to ensure a specific ordering of events. In general, probabilistic fault injection is easier to develop and use within a model. However, probabilistic injection requires more runs to ensure coverage of all possible event sequences. Hence it may be more expensive in terms of time to run the simulation and time to analyze the results. Deterministic fault injection is in general more expensive to develop because faults may have to be generated based on the local state of individual computations. It will result in the generation of all requested sequences of events in a minimal amount of simulation time. However, the deterministic approach will only generate those sequences of events desired. There may be an equally important sequence of events that is not generated because the analyst has not specified its inclusion.

6.4.3.2 Summary

The system designers guidebook demonstrates how failures in a distributed environment may be modeled using Path Pascal, a process oriented simulation language. It is useful to summarize Path Pascal's strengths and weaknesses in terms of our previous discussion on the requirements for a simulation language. Path Pascal does support multiple concurrent activities through its "process" construct. This allows the modeling of multiple sites, each of which has multiple applications executing concurrently. Path expressions provide a means for controlling shared data. If the state of a process' progress is a shared resource (e.g., encapsulated within an object), it may be accessed by multiple processes. Further if processes are divided into system processes and application processes, each of which executes a disjoint set of operations on the state information, the proper interleaving of the operations will allow the progress of a process and its state changes to be controlled. Because the language has the full descriptive capacity of a programming language and because it is process-oriented, the intricacies and side effects of a failure may be captured.

There are a number of deficiencies in the language for our purpose. Processes may be easily created dynamically, but there is no language construct for destroying them. Process destruction may be achieved by manipulating the heap from which they are allocated, but this is tricky and is discouraged. A desirable mechanism is one that simultaneously interrupts all processes of a given class (e.g., those executing on a specified processor) in order to simulate site failures. Unfortunately, there is no relation (e.g., hierarchy or classes) between processes, and there is no way of instantaneously interrupting a process.

Path expressions are intended for controlling the access to shared data by multiple processes. As such, path expressions are schedulers. However, it is difficult to use path expressions for scheduling processes for certain types of condition synchronization [ANDR83]. The way that the state of sites is disseminated when a failure occurs demonstrates one kind of condition synchronization between processes. However, often one may wish to express the states a process can go through as a form of condition synchronization with itself. For example, an application process invoking a transaction has the following specification: execute begin transaction once, followed by some number i, 0 <= i <= n, of object operations, and concluded with one abort or one end transaction. Path expressions cannot solve this problem; ad hoc solutions are required.

CHAPTER 7

PERFORMANCE EVALUATION OF THE ZEUS SYSTEM

This chapter describes the approach followed in the performance evaluation of the Zeus system. The goal of this performance evaluation work was to:

(1) Develop and illustrate modeling of recovery mechanisms and faults in distributed systems;

(2) Illustrate how to measure differential cost of introducing recovery mechanisms into distributed system designs. The performance evaluations focus on measuring the differential cost of introducing recovery mechanisms in terms of degradation in response times and throughputs of various job classes;

(3) Illustrate approaches for comparing various design options of a recovery mechanism for an application environment's fault characteristics and then selecting an option based on the results of such comparisons;

(4) Evaluate some commit protocols under various workloads and fault characteristics of the operating environment.

The guidebook presents a detailed description of how to model failures and recovery mechanisms in a distributed system using PAWS and Path Pascal. The kinds of failures considered are site crashes, disk crashes, link failures, message loss, and duplication of messages. The recovery mechanisms considered are commit protocols, reliable remote procedure calls, atomic actions, stable storage, careful replacement, object replication, checkpointing, and rollback.

The overhead introduced by a recovery mechanism is an important evaluation criterion; in order to measure this the simulation models without both the recovery mechanism and the system failures are executed, next the models with recovery ɘchanisms but without system failures are executed, and finally the models containing both the recovery mechanisms and the failures are executed. The degradation of performance from the first to the second evaluation indicates the effects of overhead because of introducing recovery mechanisms during the normal operations on the performance. The degradation of performance from the second to the third evaluation indicates the effect of overhead in taking recovery actions due to the failures conditions introduced in the system. This depends on the rate of fault injection.

PERFORMANCE EVALUATION OF THE ZEUS SYSTEM

The comparative evaluation of various design options of a recovery mechanism is important in selecting one of the options for a system design depending on the failure characteristics of its application environment and the characteristics of the jobs executed by the system. There are two parts of the comparative evaluation. The first part consists of evaluating each design option of a given set of failure rates in the system, starting with no failure case. The second part consists of determining the effect of these options on different job categories in the system. In the next section we describe a set of generic job categories in the system. The models are executed with a workload consisting of a variety of such jobs. The performance measures are collected for each job class. This helps in determining which job classes are more sensitive to the the various design options and under what kind of failure environments. Thus, given certain application system along with its job mix characteristics and failure environment, the designer can determine which option is most suitable for implementation.

As an example to illustrate these ideas we have performed comparison of the Presumed Abort vs. Presumed Commit protocol, and one-phase vs. two phase commit option. These evaluation results are presented in the last section of this chapter.

7.1 Model Overview

As described in the previous chapter, there are three components of a performance model -- environment, system structure, and workload. The environment captures the standard hardware and software as well as the effect of the physical environment. The Zeus environment included the following: configuration of the system, the performance attributes of its components, and operational conditions such as failures and their rates. The system structure captures the architecture of the parts of the model, both hardware and software, that are being analyzed. For example, the Zeus object managers with their consistency and recovery mechanisms are part of the system structure as are the command and control object managers. Finally, the workload captures the pattern and frequency of usage of various resources in the system as derived from the execution of the application systems. This includes the definition of the classes of command and control jobs. The components of the model used for this effort are described below.

7.1.1 Model Environment

The model's environment consisted of the system configuration and fault injection function. The system configuration consisted of seven sites interconnected by a local area network. Rather than model the intricacies of transmission on a local network (e.g., link control, medium access control, physical control, etc.), a delay with an exponential distribution that approximated the time of an end -to-end message was used. The hardware configuration at each site was identical. It consisted of one cpu and five disks. Two of the disks were configured to be a stable disk (i.e., the contents of the two physical disks were identical). A central server model was used, with a process requesting CPU usage and then disk usage.

All failures were assumed to be clean. The types of faults injected were site crashes and disks. It was assumed that a disk crash caused the storage medium to be corrupted and resulted in the database being reconstructed using some combination of restoration from an archival version and processing based on a log. The failure rate was varied from no failures, to a few failures, and finally to a couple of order of magnitudes increase in the number of failures. This provided a base case of operation in a fault free environment to compare with the expected case of a few faults and an extreme case of many faults.

7.1.2 Model System Structure

The system structure consisted of a number of object managers assigned to different sites and a number of transactions that could be initiated from different sites. Each site had a process/transaction manager to handle operations such as begin transaction, end transaction, and abort transaction. The object managers that performed C2 operations were instantiations of the generic object manager for exemplary C2 objects. Each object manager performed operations such as concurrency control, commit processing, and transaction undo. The amount of time to do an object manager specific operation was determined based on the number of objects accessed by an operation (see workload discussion). In addition, each site had the equivalent of the operating system support for providing transparency of object and object manager location. The details of these object managers and functions are contained in the appendix of the system designers guidebook.

Table 7-1 describes the configuration of the C2 object managers for the performance evaluation. For each object manager the following information is listed: the name of the object manger (e.g., the type of object managed), the number of instances of objects, the sites where an instance of the object manager exists, and whether or not instances of objects that are managed at multiple sites are copies that are maintained with strong consistency. The names of the sites have the following meaning: TACC is tactical air command center, CRC is control reporting center, AS1 is air squadron 1, and AS2 is air squadron 2. There are three additional sites that have none of the listed C2 object managers. They are FACP1, FACP2, and FACP3 (forward area control posts). Transactions may originate from any of the sites.

Table 7-1. C2 Object Manager Configuration

Object Manager	Number Objects	Sites	Replicated
Intelligence	80	TACC, CRC	yes
N.vigation	80	TACC, CRC	yes
Supplies	80	AS1, AS2	no
Mission Plans	120	TACC, CRC, AS1, AS2	no
Squadron	40	TACC, AS1, AS2	no
Weather	80	TACC, CRC	no

7.1.3 Model Workload

In evaluating the performance of the Zeus design, we were interested in examining the characteristics of the system with a wide variety of different job types. We were particularly interested in the effect of the different recovery mechanism design options on the performance of different job classes. However, since no application software was available, a number of generic scenarios were defined in terms of several performance-affecting attributes. Four job attributes and two possible values for each attribute were defined as follows:

- o Duration (short or long) - The total number of type manager operations executed by a scenario. The difference between "short" and "long" scenarios is about an order of magnitude.
- o Number of Objects Accessed (few or many) - The total number of objects which are either read, written, or read and written by a scenario. The difference in magnitude between "few" and "many" object accesses is about an order of magnitude.
- o R/W Ratio (R/O or update) - This indicates whether or not the scenario does any update operations on ANY of the objects that is accesses. R/O jobs do not do any update operations whereas "update" jobs do at least one.
- o Object Distribution (single- or multi-site) - If all the objects accessed by a job reside on a single host (not necessarily the same one that the scenario is running on), then the value of this attribute is "single-site", otherwise, it is "multi-site."

Of the sixteen possible generic jobs classes that may be obtained by substituting values for these four attributes, eight were chosen based on information about existing C2 applications. Table 7-2 summarizes the attributes of these eight jobs and defines an instruction mix distribution for them. The percentage figures in the job mix column indicate the percentage of the total number of jobs resident in the system at any given time (after a steady-state has been reached).

Table 7-2. Job Mix Description

Job Number	Job Mix (%)	Duration	# Objects Accessed	R/W Ratio	Object Distrib.
1	15	short	few	R/O	multi-site
2	15	short	few	R/O	single-site
3	15	short	few	update	single-site
4	15	short	few	update	multi-site
5	20	short	many	update	multi-site
6	5	long	few	update	multi-site
7	10	long	few	update	single-site
8	5	long	many	update	multi-site

By way of justification for the job mix given here, notice the following things:

- 80% of the concurrently executing jobs are short - that is, they perform relatively few operations,

- 75% of the jobs have a relatively small working set (access only a few objects),

- Many of the jobs (30%) are read-only.

Although this method of selecting an example workload may seem somewhat ad hoc, it is expected that it will provide valuable performance data that is sufficiently accurate to guide the design process. More importantly for our present purposes, it provides a concrete example of the use of instruction mixes in real design situations.

For each job description in the mix, a number of exemplary jobs were required. Synthetic jobs were used since we were modeling a pre-operational systems. These are artificial jobs for which the resource usage is similar to the expected characteristics of some future real jobs or to existing jobs being run on other systems.

Synthetic jobs are easier to obtain than the real applications because they summarize the resource utilization of the jobs that they are characterizing. Local CPU usage, for example, is usually represented by a simple idle loop in a synthetic job and remote requests are abstracted so that the parameters of the calls are simplified or left out entirely.

As an example of a synthetic user-level scenario from the Zeus performance analysis, consider the following job:

```
Begin Scenario
     Read Intelligence
     Read Navigation
     Read Weather
     Read Mission-Plan
     Computation
     Update Mission-Plan
End Scenario
```

It is assumed that "Intelligence", "Navigation", "Weather", and "Mission-Plan" are objects (or groups of objects) in the command and control system. The semantics of the scenario is meant to resemble a so-called "Mission Control" job which is a typical (although much simplified) job being run on other C2 systems. The scenario reads the appropriate data, does some local computation to determine how the plan of some in-progress mission should be changed, and then updates that mission plan.

Notice how this synthetic job represents the basic performance-affecting features of the scenario in a very stylized and simplified way. The meaning and functionality of the local computation is not specified and neither are the parameters of the remote calls. A complete description of the jobs is contained in the appendix of the system designers guidebook.

7.2 Goals of the Example Evaluation

The example evaluation is performed with the objective of evaluating certain design options for commit protocols. Specifically, in this evaluation we investigate the Presume Commit vs. Presume Abort option, and one phase commit vs. two phase commit option.

The Presume Abort protocol implies that in the absence of any information about a transaction's commitment at its coordinator, it is presumed that the transaction was aborted. This means that in case of commiting a transaction, the coordinator must keep the transaction's commit status information until it is certain that no status queries for that transaction would be received in future. Analogously, the Presumed Commit protocol implies that in the absence of commit information, a transaction is presumed to have committed. Thus, when a transaction is aborted, the coordinator must maintain its abort status until it is certain that no more status queries for that transaction would be received in future. In a fault-free environment where most of the transaction get committed, it looks more attractive to follow the Presumed Commit protocol; this avoids synchronous disk writes for a large number of transactions. On the other side, if a system encounters large number of failures that lead to the majority of transactions to be aborted, it looks more attractive to use the Presumed Abort protocol. One can observe that as the failure rate in a system is increased starting with a fault-free system, there exists a point of inflexion where it is more advantageous to use the Presume Abort protocol. This point of inflexion can be obtained by executing the model with varying rates of fault injection.

Another design option that we investigated is one phase vs. two phase commitment. The one phase commit protocol implies that in response to every update operation on an object, its Type Manager creates a new commit pending version of the object on the stable storage. The object remains in the commit pending state until the Type Manager receives the decision about the commitment/abortion of the client transaction. The commit pending state implies that the object can not be used by other clients until the commit decision is received from the coordinator. A coordinator failure while an object is in the commit pending state will cause the object to remain unavailable to other clients. The period during which an object is in the commit pending state is called its in-doubt period. The two phase commit protocol tends to reduce this window of vulnerability. In this protocol, each update operation creates an uncommitted version of the object. At the end of a transaction, its coordinator executes a protocol which first attempts to make every object accessed by that transaction commit pending. After this phase, it makes the commit/abort decision. The two phase commit protocol requires additional messages, but it tends to reduce the window of vulnerability.

Obviously, the one phase commit protocol is preferred if there are few failures in the system; however, in an environment where the failure rates are high, it is more desirable to use the two phase commit protocol. The two phase commit protocol introduces overheads in terms of extra messages and disk I/Os. These overheads may not be justfiable for short transactions. In our example evaluations we investigate how to determine which option would be most suitable for a given application.

7.3 Summary of the Evaluation Data

To do the evaluation we chose to hold the hardware architecture constant, the software object manager to processor allocation constant, and the workload generation constant. A set of five fault injection rates were defined. They varied from no faults to 12.8 fault per 100 seconds. The system structure was varied by using different commit protocols between the generic object manager and process/transaction manager. Four different commit protocols were modeled -- one phase presumed abort, two phase presumed abort, one phase presumed commit, and two phase presumed commit. The model was run for each protocol for each failure rate until 1000 transactions had successfully committed. Some of the highlights of the analysis are summarized here.

Figure 7-1 shows the effect of a commit protocol on the throughput of the overall workload as the failure rate increases. The overall transaction throughput summary demonstrates the performance degradation due to the increasing occurrence of faults. There are three main points to note -- the effect of presumed abort versus presumed commit, of two phase versus one phase, and of timeout periods. Presumed commit protocols outperform presumed abort protocols for low fault rates as expected. But for one phase protocols, the presumed abort protocol outperforms the presumed commit protocol when the fault rate execeeds 5 faults/100 seconds. This indicates that it may be desirable to have an adaptive commit alogrithm that uses a presumed commit protocol when the environment is not faulty and switchs to a presumed abort protocol when a fault rate surpasses a given threshold.

One phase protocols outperform two phase protocols. This is not surprising for environments with a low fault rate. It is somewhat surprising for environments with a very high fault rate. This can be explained by two observations. First, the slope of the curve of a two phase protocol tends to decrease more rapidly than that of one phase protocols indicating that there may exist some fault rate at which two phase protocols do indeed outperform one phase protocols. Second, the model of the time duration of a device failure is unrealistically short. This is due to the excessive time and resources that would be required to run a simulation that accurately modelled a device's downtime. The effect on a one phase commit protocol of a longer downtime is to increase the size of the window of vulnerability (or in-doubt period) of a server to the failure of a coordinator. This would increase the period during which a set of objects would be indefinitely blocked, thereby reducing the potential system concurrency and therefore throughput.

The timeout period which an object manager holds a lock for a transaction using a two phase commit protocol can unduly effect the throughput. A short period may result in many transactions being aborted unnecessarily. This is explained as follows. As the multiprogramming level increases, the concurrency level of the object managers increases. When the objects are reliably manipulated there are extra disk accesses to store stable states. The increased number of disk accesses results in a bottleneck at the stable storage device. This results in a longer period of time that a transaction waits for a response from an object manager and that an object manager holds a lock for a transaction. As the stable storage request queue grows, the number of aborted transactions increases. Note that this does not happen for one phase commit protocols because objects are placed in a commit pending state following their first access, nor does it happen for environments with a high fault rate because the multiprogramming level is reduced. This problem can be avoided for a two phase protocol, if either the multiprogramming level is reduced or the timeout period increased. This explains why the throughput of the two phase presumed abort run with no faults is as low as the few faults run.

Figures 7-2 through 7-5 show the effect of the commit protocol with varying fault rates on the workload mix. It demonstrates that as the fault rate increases shorter transactions tend to dominate the mix of successful transactions. Short appears not to be sensitive to the number of objects accessed, but to the number of operations on the set of objects. Further, the overhead of a two phase protocol did not seem to be warranted for short transactions under any fault rate.

In this chapter we have presented the goals of the example system evaluation, the description of the example system, and the evaluation of some commit protocols under various failure characteristics of the operating environment. A detailed description of this evaluation is presented in the guidebook.

Figure 7-1. Effect of failure rate and commit protocol on throughput.

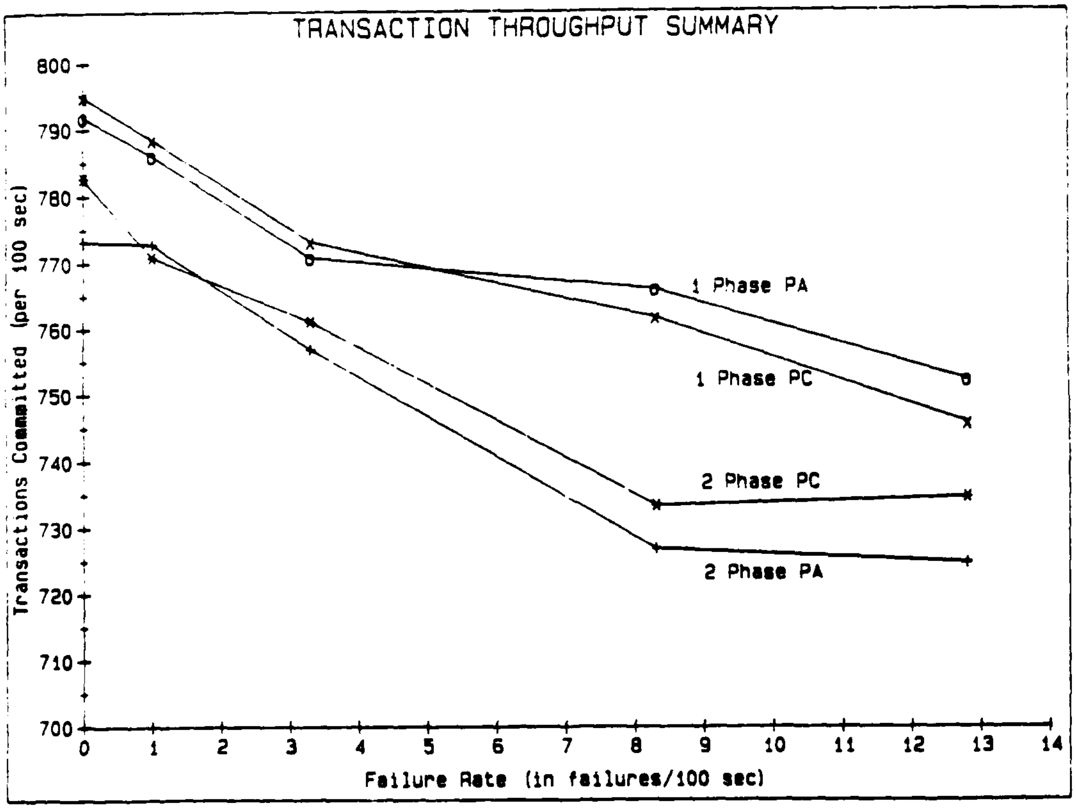
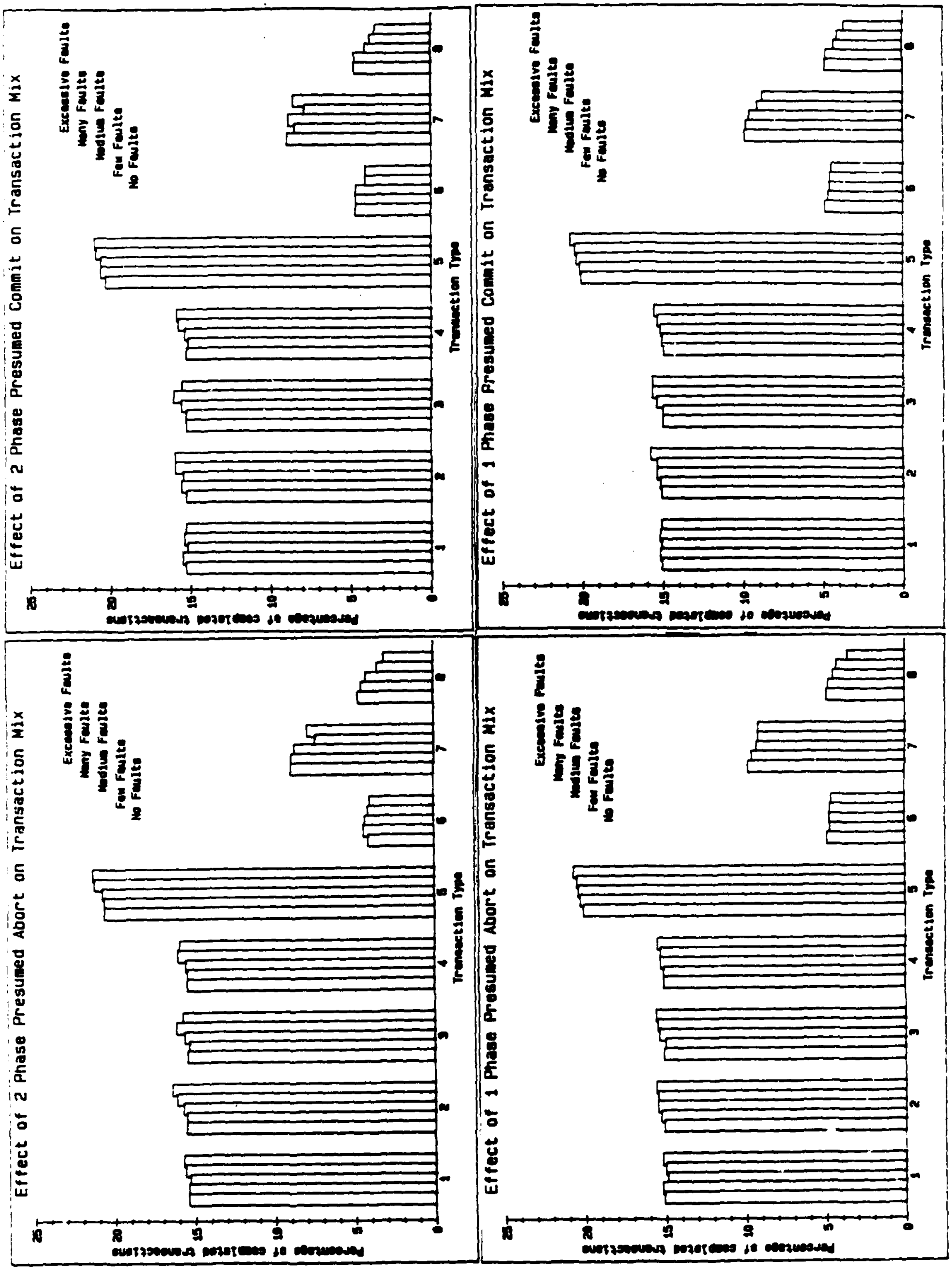


Figure 7-2 - 7-5. Effect of failure rate and commit protocol on job mix.



CHAPTER 8

FUTURE DIRECTIONS

The long range goal of research on distributed system recovery mechanisms is to develop a methodology that allows a system designer to select an appropriate set of recovery mechanisms for a given system environment and workload. The work done for this contract has created a foundation for this goal. This report has presented a summary of that work. The details of the work are reported in the system designers guidebook. There are a number of areas which warrant further research. In this chapter, we discuss four areas -- system structuring, analysis and validation techniques, design specifications, and a designer's workbench.

8.1 System Structuring

System structuring topics to explore include the following: advanced studies in reliability techniques, reliable process oriented systems, the impact of failures on system security, implementation issues for object oriented systems, and distributed programming environments. Advanced studies into reliability techniques can be conducted in either an application specific or application independent manner. An application specific approach examines the requirements of a specific application and produces results that are very appropriate for a specific class of applications. There is the potential drawback that the results may not be generalizable to other classes of applications. The goal of research in this area is to develop techniques that provide increases in the performance and reliability of recovery mechanisms for distributed command and control applications. The development of non-serializable transaction processing techniques that take into account the semantics of command and control operations is a fruitful area for exploration. This research may be pursued either through detailed simulation or experimental evaluation.

Research into generic reliability techniques has the goal of producing a handbook of distributed system recovery mechanisms. The handbook would describe the performance and reliability of generic recovery mechanisms, and identify what mechanisms are appropriate for what kind of applications. There are two avenues of investigation. The first is to develop new algorithms, analyze their performance and reliability attributes, and determine for what applications they are useful. The development of a general theory of non-serializable transaction processing based on the semantics of operations and/or probabilistic decision making is an example of this kind of exploration. Algorithms and strategies to dynamically partition, assign, and reconfigure objects that result in increases in performance and reliability are another example. The second is to do a detailed study of existing mechanisms. A study could compare for the management of recoverable objects different techniques such as differential files, logs, and careful replacement; for reliable transactions the effect of concurrency, deadlock, timeouts, and failures in conjunction with commit protocols; and for replication management the effect of different levels of replication and network partitions on consistency and recovery techniques. These explorations may be done either through the use of detailed simulation or experimental evaluation.

The work done on this contract has focused on object oriented system designs. Many of the existing systems have been developed with a process oriented structure. There are two possible areas of exploration. The first considers the application of object oriented design and recovery to real time systems, a set of applications that have been traditionally developed using process oriented techniques. The second considers existing and new recovery techniques for reliable process oriented systems. The techniques can be explored through detailed simulation or experimental evaluation.

Security is one area of operating system functions that this work did not explore. Existing security policies are based on centralized management techniques; it is assumed that a system is either running or stopped. But in a distributed system it is possible for some system components to fail and for the rest of the system to continue operating. The question arises as to what is the impact of failures on security.

Zeus, an object oriented design of a distributed system, was developed and used for modeling in this contract. It was demonstrated that object orientation provides a number of advantages for recovery. There are a number of unanswered questions about how a reliable object-oriented distributed system should be implemented. There are two kinds of problems associated with a development effort for building a Zeus-like system. The first aspect is related to certain generic problems in implementing objects. Examples of these questions include how to efficiently implement functions, such as transparency of location, replication, failures, and concurrency. The second kind of problems are related to the software development environment such as the selection of appropriate operating system kernel, programming language(s) and tools such as compilers, linkers, loaders, debuggers, etc. The software selection is a difficult task because of the small number of languages and tools that exist for distributed environments. This is a critical problem because the failure to obtain the proper tools may require additional effort in building such tools. Building a Zeus-like system on some commercially available workstation along with its operating system such as UNIX(1) may require tailoring of the kernel functions to facilitate efficient implementation of recovery mechanisms. Such modifications to the host operating system are engineering research problems which need to be investigated.

(1) UNIX is a registered trademark of Bell Laboratories

Distributed programs are difficult to develop. One approach to easing their development is to use an object oriented approach combined with a run time environment that provides transparency as described above. A complementary approach is to integrate high level non-procedural constructs into an object oriented environment. This further eases the difficulty and time required for developing distributed programs, resulting in an increase in programmer productivity. There are several research topics associated with such a distributed programming environment, including: non-procedural language constructs, translation of non-procedural constructs that may include conditionals into sequences of operations on objects, and the partitioning and assignment of objects in a distributed environment.

8.2 Analysis and Validation

The design evaluation methods cover system attributes such as reliability, performance, and functional correctness. There are several directions for future research and development in the area of design evaluation methods.

In the area of performance modeling, there is a need to investigate analytical models, possibly based on Markov chains, of distributed system recovery mechanisms and to develop analytical techniques to predict the performance of reliable and survivable distributed systems using these models. Modeling of checkpointing, rollback, commit protocols, and replication management protocols should be included in this effort. An interesting area of investigaion could be development of analytical models of protocols for replication management under weak consistency requirements. The development of analytical models that faciliate both performance and reliability evaluations and their interactions in a fault tolerant system is an important research area. The performance evaluation of the example system in this effort is based on simulations using PAWS. Our experience in this effort indicates that it is desirable to have an advanced simulation language that provides convenient mechanisms for modeling faults and their effects in a distributed system. For example, a language construct that stops progress of all computations associated with a failed component would be useful.

In this effort, the work related to the application of program verification techniques focused on the construction of recoverable objects at a single site. The verfication of protocols for constructing distributed recoverable objects using program verification techniques such as Gypsy is not completely addressed in this effort. The problem of protocol verification needs a significant level of additional work. In this contract we propose an approach using Finite State Machines and interval logic to reason about the correctness of such protocols. In the system designers guidebook this method is developed and illustrated using an example. This method appears promising because it is simpler than program verification techniques. Efforts are needed to develop a formal theory for the verification of distributed system recovery protocols using this method. It should then be possible to build some automated tools for applying this method.

8.3 Formal Methods for Design Definition

One of the important features desired in a design definition langauge for reliable systems is an exception condition handling model. Concurrent System Definition Language, which does not have this feature, can be extended to include an exceptional handling model. Formal specification of functional requirements along with their performance and reliability characteristics is important during the various design phases. It would be interesting to investigate such a specification langauge in the context of Ada or CSDL.

8.4 System Designers Workbench

The system designers guidebook presents a set of techniques and tools for evaluating reliable distributed system designs. These tools include PAWS for performance evaluation, NetRAT for reliability evaluation, Gypsy for formal verification, and Path Pascal for functional simulations. One can envision a system which integrates these tools into a designers workbench system which facilitates the convenient application of these tools to distributed system design expressed in some formal design langauge. This workbench would automatically translate a design expressed in the design langauge to the appropriate evaluation model required for an evaluation tool. It would also guide the designer during the analysis procedure and ask questions regarding any information that is necessary for evaluations but not specified in the design.

8.5 Recommendations

Distributed processing research is in a state of flux. There are an abundant number of concepts about how to develop systems and what functions system should contain. However, there is a shortage of experience in applying these concepts in the actual development of systems. The insight that one can gain from the experimental evaluation of a system differs dramatically from what one can determine from modeling and analysis. The data and subsequent insight that is needed to make significant progress can be fostered only through the actual observation of a phenomena. Therefore, it is strongly recommended that work be continued on the general topic of system structuring using experimental evaluation. To better ensure the relevance of future results, we recommend an approach with two thrusts that may require the participation of multiple organizations. One thrust examines command and control applications in detail, resulting in a detailed design of a demonstrable subset of a command and control application. The other thrust pursues the experimental evaluation of generic reliability techniques using the above application as a test vehicle. The implementation should provide an object based distributed operating system, the use of non-procedural language constructs, and tools that aid in the partitioning and assignment of objects in a distributed environment. The experimental evaluation should provide data as to how the recovery mechanisms, object-oriented operating systems, and non-procedural language constructs support distributed command and control applications.

REFERENCES

[ADRI82] Adrion, W., et al., "Validation, Verification, and Testing of Computer Software," Computing Surveys, 14(2), pp. 159-192, June 1982.

[ANDR83] Andrews, G., and Schneider, F., "Concepts and Notations for Concurrent Programming," Computing Surveys, 15(1), pp. 3-43, March 1983.

[BALT81] Balter, R., "Selection of a Commitment and Recovery Mechanism for a Distributed Transactional System," Proc. First Symposium on Reliability in Distributed Software and Database Systems, 1981, pp. 21-26.

[BERN81] Bernstein, P.A., Nathan Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2, June 1981, pp.185-222.

[BROW75] Browne, J. C., K. M. Chandy, R. M. Brown, T. W. Keller, D. F. Towsley, and C. W. Dissly, "Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems," Proceedings of IEEE, Vol. 63, No. 6, 1975.

[COHE75] Cohen, Ellis, David Jefferson, "Protection in the Hydra Operating System," Fifth Symposium on Operating Systems Principles, 1975, pp. 141-160.

[DAVI73] Davies, C.T., "Recovery Semantics for a DB/DC System," ACM National Conference, 1973, pp.136-146.

[DoD83] United States Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A, 1983.

[ESWA76] Eswaran, K.P., et. al., "The Notion of Consistency and Predicate Locks in Database Systems," Communications of the ACM, 19, 11, November 1976, pp.624-633.

[FRAN83a] E. Frankowski, W. Wood and R. Orgass, "Concurrent System Definition Language, Volume One, Description Language," Tech. Report CTC-R-83-17; 8213, Honeywell Computer Sciences Center, September 1983.

[GERH80] Gerhart, S.L., et al., "An Overview of AFFIRM: A Specification and Verification System, In Proceedings IFIP 80, pages 343-348, October 1980.

[GIFF79] Gifford, D. K., "Weighted Voting for Replicated Data," Seventh Symposium on Operating Systems Principles, 1979, pp.150-162.

[GOOD78] Good, D. I., Cohen, R. M., Hoch, C. G., Hunter, L. W., Hare, D. F., "Report on the Language Gypsy, Version 2.0," Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September 1978.

[GOOD79] Good, D. I., Cohen, R. M., Keaton-Williams, J., "Principles of Proving Concurrent Programs in Gypsy," Proceeding of the 6th ACM Symposium on the Principles of Programming Languages, 1979, pp. 42-52.

[GOOD82a] Good, D. I., "The Proof of a Distributed System in Gypsy," in Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar - M. J. Elphick)Ed._. September 1982. Also Technical Report #30, Insititute for Computing Science, The University of Texas at Austin.

[GOOD82b] Good, D. I., Siebert, A E., Smith, L. M., "OSIS Message Flow Modulator - Status Report," Internal Note #36A, Institute for Computing Science, The University of Texas at Austin.

[GREI77] Greif, I., "A Language for Formal Specification,"Communication of the ACM, December 1977, pp. 931-935.

[GRNA80] Grnarov, A., Kleinrock, L. Gerla, M., "A New Algorithm for Symbolic Reliability Analysis of Computer Communication Networks," Proc. Pacific Telecommunications Conference, January 1980, pp. 1-9.

[GRNA81] Grnarov, A., Gerla, M., "Multi-terminal Reliability Analysis of Distributed Processing Systems," Proc. 1981 International Conf. on Parallel Processing, August 1981, pp. 79-86.

[HALP83] J. Halpern, Z. Manna and B. Moszkowski, "A Hardware Semantics Based on Temporal Intervals," Tech. Report STAN-CS-83-963, Stanford University, March 1983.

[HORN74] Horning, J.J., et. al., "A Program Structure for Error Detection and Recovery," Computer Science, Springer Verlag Lecture Notes in "Operating Systems Techniques", Volume 16, .

[IRA83] Information Research Associates, Performance Analyst's Workbench System - Introduction and Technical Summary, Information Research Associates, Austin, Tx., July 1983.

[KEMM80] Kemmerer, R., "FDM - A Specification and Verification Methodology," In Proc. 3rd Seminar on the Department of Defense Computer Security Initiative Program, Nat. Bur. Stand., November 1980.

[KIM79] Kim, K.H., "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," First International Conference on Distributed Processing, 1979, pp.284-295.

[KOBA78] Kobayashi, H., Modeling and Analysis: An Introduction to Performance Evaluation Methodology, Addison Wesley, 1978.

[KOHL81] Kohler, W. H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

[KUNG83] Kung, H.T., John T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, pp.231-226.

[LAMP81a] Lampson, Butler W., "Atomic Transactions," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siegert, Springer-Verlag, 1981, pp.246-265.

[LAMP81b] Lampson, Butler W., "Remote Procedure Calls," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siegert, Springer-Verlag, 1981, pp.365-370.

[LISK82a] Liskov, B., "On Linguistic Support for Distributed Programs," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp.203-210.

[LISK82b] Liskov, B., Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Ninth Annual Symposium on Principles of Programming Languages, January 1982, pp.7-19.

[MISR81] Misra, j., Chandy, K. M., "Proof of Networks of Processes," IEEE Transaction on Software Engineering, July 1981, pp. 417-426.

[MOHA83] Mohan, C., Lindsay, B., "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, August 1983, pp. 76-88.

[MOSS81] Moss, J. Elliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA 02139, April 1981, pp..

[MOSZ83] B. Moszkowski, "A Temporal Logic for Multi-Level Reasoning about Hardware," Computer Hardware Description Languages and their Application, T. Uehara and M. Barbacci, eds, North-Holland Pub. Co., 1983.

[NELS81] Nelson, B.J., "Remote Procedure Call," Technical Report-Department of Computer Science, Carnegie-Mellon University, May 1981, pp..

[REED78] Reed, D.P., "Naming and Synchronization in Decentralized Computer Systems," Technical Report MIT/LCS/TR205, September 1978, pp..

[RIES82] Ries, Daniel, R., Gordon C. Smith, "Nested Transactions in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No.3, May, 1982, pp.167-172.

[ROBI75] Robinson, L., Levitt, K. N., Neumann, P. G., Saxena, A. R., "On Attaining Reliable Software for a Secure Operating System," In International Conference on Reliable Software, pp.267-284, IEEE, April 1975.

[RUSS80] Russell, D.L., "State Restoration in Systems of Communicating Processes," IEEE Transactions on Software Engineering, Vol SE-6, No.2, March 1980, pp.183-194.

[SALT81] Saltzer, J.H., Reed, D.P., Clark, D.D., "End-To-End Arguments in System Design," Second International Conference on Distributed Computing Systems, 1981, pp.509-512.

[SCHA83] Schantz, R., et al., "Cronus: A Distributed Operating System - Interim Technical Report No. 2, "RADC Report No. 5261, February 1983.

[SHRI81] Shrivastava, S.K., "Structuring Distributed Systems for Recoverability and Crash Resistance," IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981, pp.436-447.

[SHRI82a] Shrivastava, S.K., Panzieri, F., "The Design of a Reliable Remote Procedure Call Mechanism," July 1982, pp.692-697.

[STON81] Stonebraker, Michael, "Operating System Support for Database Systems," Communications of the ACM, July 1981, pp.412-418.

[SVOB81] Svobodova, L., "A Reliable Object-Oriented Data Repository for a Distributed Computer," Eighth Operating Systems Principles Conference, 1981, pp. 47-58.

[THOM79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp.180-209.

[WALK83] Walker, B., et al., "The LOCUS Distributed Operating System," Ninth ACM Symposium on Operating System Principles, October 1983, pp. 49-70.

[WANG83] Wang, P.S., Tripathi, A.R. "An Algorithm for Network Reliability Computations," Research Report, Honeywell Computer Sciences Center, Bloomington, MN 55420.

Appendix I

CSDL: Concurrent System Definition Language

William T. Wood
Elaine N. Frankowski

April 1984

Honeywell Corporate Computer Science Center

CSDL: CONCURRENT SYSTEM DEFINITION LANGUAGE

1.1 INTRODUCTION

Concurrent system design is an engineering activity which requires software engineering technology comprising a design methodology, design methods and languages, and tools to automate its procedures. This chapter presents methodological principles and linguistic support for engineering constructively correct concurrent system designs. The Concurrent System Definition Language (CSDL) is based on a formal model of computation, giving it both rigorous foundations to support the less formal creative process, and mechanical means, such as mapping functions, of supporting system engineering. CSDL integrates software development techniques which have not been combined before (data abstraction, information hiding, temporal logic, Dijkstra's guarded commands) allowing designers to create and reason about data, algorithms, and communication architecture. CSDL contains both a description and a specification language, permitting designers to carry out the entire design process in the same syntactic and semantic environment.

CSDL is a collection of seasoned techniques, mechanisms and language constructs that have not been combined before. It is rewarding to discover that many significant individual contributions to software design can be integrated into a single system without major clashes, and that they do function in an integrated way to provide the desired reasoning vehicle for constructing verifiably correct designs. It is also rewarding to discover that even when CSDL is used informally, it succeeds in helping designers produce more robust designs and have more control over the design process.

CSDL succeeds in managing design complexity by encouraging, almost forcing, an architectural view of systems. This architectural view is compatible with both top-down and bottom-up design styles; in each case a collection of system elements is "hooked together" to construct a system that satisfies a specification. CSDL also succeeds as a means of producing implementation blueprints because it has constructs for expressing data structure, procedural behavior and communication architecture.

A major part of detailed design of the Zeus operating system was done using CSDL. These designs are presented Volume II of this guidebook.

This chapter presents CSDL's computational model and model of system architecture, its methodology and language constructs, an example using it and some possible directions for its enhancement.

CSDL: CONCURRENT SYSTEM DEFINITION LANGUAGE

Some requirements we identified for a software engineering package are based on our recognition of software engineering as an instance of engineering in general, and our understanding of the engineering problem. They are:

1. A well-defined theoretical model of computing to support a rational, creative software design and the development process. The model constitutes scientific underpinnings that define the way engineers think about both the software product and the methods and components used to build it.

2. Technical methods for use by individual developers in engineering a partial or complete software system design. These are methods, such as data abstraction, which the individual practitioner applies to the design task, not methods such as version control or configuration control that are applied to managing whole projects.

3. Support for creative freedom and realistic analysis of correctness, feasibility, and economy of alternatives. The models, methods and languages the project develops must accommodate both the creative ideas that experienced designers get, and the analytic methods they use to determine the effects of their ideas.

4. Techniques to manage and reduce the complexity of the design process, the resulting design and the design document. The technical methods must include facilities for decomposing the system design task so that it becomes intellectually manageable.

5. A design language that presents a software system design in the form of a description explaining how the system is built up from smaller pieces, and how those pieces are connected, along with a specification that explicates the expected observable behavior of the system, its parts, and the mechanisms that connect them. A system's description constitutes the blueprint of components and interconnections from which it is built; its specification presents the relevant properties of the device as a whole, of its parts, and of the interconnection mechanisms.

6. A design notation that expresses the product's specifications and description in the terms of the implementing technology. A detailed design is solution oriented, so it is expressed using statements about the processes, input and output values, algorithms and memory of software technology, not the user interfaces, applications packages, sensors, actuators, or transducers of user requirements technology.

Other requirements for the CSDL software engineering package arise from the fact that it is meant to be used by people. They include:

- The models and languages must have intuitive appeal to software designers and programmers.

- The methods must be automatable, so that machines, not humans, can deal with detail.

These requirements led to the development or adoption of:

1. A formal model of sequential and concurrent computations.

2. A system model that characterizes the building blocks with which systems may be designed.

3. Methodological principles and guidelines that define desirable properties of the design activity, the design language and the design itself, and make procedural suggestions for carrying out the design process.

4. Technical methods essential for engineering software. They are, for example, data abstraction, procedural abstraction, Dijkstra's constructive approach, and the like.

5. A description language - a formal notation for describing how a system is built up from pieces and how those pieces are connected. Its semantics are based on the model of computation.

6. A specification langauge - a formal notation for documenting the expected behavior of a system description. Its semantics are based on the formal model.

7. Analytic methods for investigating operational properties such as performance, reliability, or security of alternative functionally correct system designs.

These elements are applied to detailed design, development phase whose work product is a design documenting a system's logical architecture, its paths of information flow, the data type of each system object and the behavior of the system and each of its modules. A detailed design expresses what will actually be implemented. Each object in the design -- module, data object, procedure, or information flow path -- will exist in the implementation, though the object's physical realization may be different from its logical design. For example, a type operation designed as a procedure may be implemented with in-line code.

The remainder of this chapter is organized as follows: Section 1.2 presents CSDL's underlying formal models of computation and system architecture. Section 1.3 explains the CSDL design methodology and presents the major linguistic features that support it. Section 1.4 presents the CSDL constructs for describing and specifying system designs. Section 1.5 is an example system design. Section 1.6 draws appropriate conclusion, outlines possible improvements and suggests direction for future work.

1.2 MODELS

Both CSDL's computational model and its system model define ways that people can think about the software they build. The computational model is a formalized, mathematical abstraction of the phenomena that an engineer manipulates when building an artifact. It is the vehicle for theoretical work. The system model is qualitatively different. It characterizes the abstract building blocks of a concurrent system design. Thus, the system model introduces conceptual constraints on the model of computation, implying that CSDL's computations will not be realized in all the ways that the computational model allows, but only in those ways which can occur on this conceptual architecture. CSDL's computational model was used in developing the language and in presenting it in [FRAN83a] and [FRAN83b]. In this section, the language is explained more informally so the reader may choose to skip Section 1.2.1 However, the system model (Section 1.2.2) is used in the remainder of this chapter.

1.2.1 Computational Models

CSDL's computational model formalizes the concepts needed to talk about the structure and observable effects of a large class of programming mechanisms. Our goal is a precise and rigorous model that expresses the essentials of the things system engineers work with simply and intuitively. Precision and rigor are necessary if the results of reasoning in the model are to be trusted. Simplicity and intuitiveness are necessary if the model is to be adopted by people whose task is to build things, not to philosophize about them.

Section 1.2.1.1 presents a model of sequential computations; Section 1.2.1.2 presents a model of concurrent computations and Section 1.2.1.3 explains system histories which are used to reason about system behavior.

1.2.1.1 Sequential Computations

Our model of sequential computations is a very conventional one based on the primitive notions of states and transitions. This model appears to be sufficient to define program semantics in terms of effects on data and parameters.

A data object x is an entity that can take on any value V(x) of a specified set of values T(x). The *state* of x at some point during its lifetime is its value V(x) at that point. Given a set X of n objects x1,...,xn, where each xi is of type Ti, the state q(X) at some point in the lifetime of X is given by the vector of values of the objects

$$q(X) = < V(x1),...,V(xn) >$$

at that point. X is called an *object space*, and the set Q(X) of all such vectors is called the *state space* of X.

A single terminating sequential program P defined over an object space X effects a state transition on X in that it is invoked with X in one state q(X) and terminates with X in some state q'(X). The effects of P may be expressed by a binary relation [P] over Q(X). The interpretation of this relation is that the pair < q(X), q'(X) > is an element of [P] if and only if P is guaranteed to halt when invoked from q(X), and q'(X) is one of the states in which P can halt when invoked from q(X). With this interpretation it follows that the domain of [P], that is, the set of all states that can be first elements of pairs in [P], is exactly the set of initial states from which termination is guaranteed.

An execution of P over X may be modeled by a sequence of states h(X,P):

$$h(X,P) = q0(X); \ldots qn(X); \ldots$$

where q0(X) is the state of X at the invocation of P. If q0(X) is an element of the domain of [P] then h(X,P) is finite, and if in addition qn(X) is the last state of h(X,P) then < q0(X), qn(X) > is an element of [P].

Different designs of a program that has a given desired effect may be distinguished by their possible execution sequences, which reflect the intermediate states a particular design will pass through while attaining the desired over-all effect.

1.2.1.2 Concurrent Computations

The sequential model does not deal with the notion of time or of an external environment with respect to which a program causes change through time. These phenomena are treated by CSDL's model of concurrent computation.

A concurrent computation is modeled as a collection of m sets of object spaces X1,...,Xm, with a program Pi defined over each Xi. The object spaces Xi may overlap or have elements in common. This occurs in a CSDL design when two programs share communication objects. Objects may move from space to space; that is, an object may "instantaneously" vanish from space Xi and appear in space Xj. This occurs in a CSDL design when one program dynamically creates a process and gives some of its objects to the newly created process. Attempts by multiple programs to operate on a shared object are nondeterministically serialized. The transfer of an object from one space to another is serialized with all other operations. With the exception of serialization of operations on shared objects, the programs Pi over the object spaces Xi proceed asynchronously and independently of one another.

Object spaces may vanish, and new ones appear throughout the lifetime of a concurrent computation. This is expressed in a CSDL design by dynamic process creation and destruction. The computation exists as long as at least one space exists. When a new space appears, its objects may all be new or, as stated above, some of them may come from an existing space. When a space containing such "borrowed" objects subsequently vanishes, the borrowed objects are returned if the "lender" still exists. Otherwise they vanish. That is, if a program dynamically creates a process, gives control of some of its objects to the new process and then destroys the process, the objects it yielded control over return to its control.

This model of concurrent computations introduces complications into the model of sequential computations. A single space X with program P can have objects disappear and appear throughout its lifetime as it creates and destroys processes. Also, objects that are shared with other spaces can appear locally to change state asynchronously with, and without being manipulated by P. That is, information enters from P's environment.

1.2.1.3 Histories

An execution sequence h(Xi,Pi) of a program Pi over object space Xi is a particular (possible) *history* of Xi. The set H = { h(X1,P1),...,h(Xn,Pn) } is a possible history of the concurrent computation. H may have some elements that appear and vanish and others that are infinite; this corresponds to some object spaces appearing and vanishing while other object spaces last forever once they are created. The only restrictions on state and rate of the various members of H are those that arise from shared and moving objects. In particular, although each history h(Xi,Pi) is totally ordered, there is only a strict partial order, *precedes*, among states in H. The precedes relation is the basis for the notion of temporal order. The precedes relation may be defined recursively as follows:

(1) Within a history h(Xi,Pi) the state qj(Xi) precedes qk(Xi) if and only if j < k.

(2) If x is shared by spaces Xi and Xj, if qm(Xi) and qm+1(Xi) are consecutive states in h(Xi,Pi) corresponding to a change in x, and if qk(Xj) and qk+1(Xj) are consecutive states in h(Xj,Pj) corresponding to the same change in x, then qm(Xi) precedes qk+1(Xj).

(3) If qp(Xi), qr(Xj), and qh(Xk) are states anywhere in the system, and if qp(Xi) precedes qr(Xj) and qr(Xj) precedes qh(Xk), then qp(Xi) precedes qh(Xk).

The precedes relation is partial rather than total because there can exist distinct states qk(Xi) and qm(Xj) neither of which precedes the other. For example, suppose Xi and Xj share x, and further suppose that Pi changes x from 2 to 3 and later Pj changes x from 3 to 4. All states of Xi up to the change from 2 to 3 precede all states of Xj after that change, and all states of Xj up to the change from 3 to 4 precede all states of Xi after the change. But the states of Xi after the change from 2 to 3 but before the change from 3 to 4 have no defined relation to the states of Xj in the same period; they are incomparable.

1.2.2 System Model

Our system model defines a conceptual architecture for concurrent systems. It limits the universe of designs for realizing a specified computation to those that can be defined within such an architecture.

A concurrent system is a collection of machines which operate concurrently and autonomously. They communicate asynchronously by passing information. Internally, a machine consists of data objects and procedures and/or subordinate machines to manipulate these objects. A machine containing only procedures constitutes a sequential locus of control. A machine containing subordinate machines constitutes several autonomous control sites. If the system's architecture is viewed as a tree, its leaves are all sequential control sites.

Machines may also contain machine pools from which machine instances may be created and destroyed as the system runs.

Systems are evolutionary. The initial system configuration is described by a distinguished machine, SYSTEM. SYSTEM may contain other machines and machine pools. Each machine that SYSTEM contains may, in turn, contain other machines and machine pools. The initial system is, then, the configuration consisting of SYSTEM, all machines it contains, and all the machines they contain. A system evolves by dynamic creation and destruction of machines from pools. Since every pool element may contain machines and machine pools, creating a new machine dynamically may, in effect, create a new subsystem.

A system's communication architecture is the set of connections among its machines. Connections are formed among active objects, objects whose values can change without being manipulated by the machine which contains them. Since machines cannot manipulate each other's objects, a communication link is set up by connecting an active object in one machine to a complementary (roughly same type, opposite direction) active object in another. The sending machine puts a value in its local active object, and that value is instantaneously transmitted to the complementary active object from which the other machine can get it by a local operation. Active objects may be connected to realize point-to-point, multi-cast, fan-in and broadcast communication architectures. Connected active objects by definition correspond to shared objects in the computational model, as described in Section 1.2.1.2.

1.3 METHODOLOGY

CSDL's methodology comprises design guidelines which are a synthesis of concepts drawn from research on software design. They offer procedural suggestions for carrying out the design activity. One practical effect of adopting design guidelines was to include features to support them in CSDL.

1.3.1 Constructive Correctness

Basically, CSDL advocates building in correctness; that is, using a system component's specification as a driver for constructing its description. The constructive design process at every level· begins with a requirements analysis. At the topmost level these specifications of requirements are usually arrived at through discussions among designers, requirements analysts, and customers. The task of designing procedures, data, and machines then uses these top level specifications as its starting point.

A procedure's functional specifications are obtained by constructing assertions that define a constraint on valid inputs and the relation between valid input and desired output. These assertions are defined over the variables global to, and the parameters of, the procedure. A procedure's behavioral specifications are obtained by constructing assertions that characterize state sequences over the global variables and parameters associated with invocations of the procedure. These assertions express properties both of individual invocations, such as time performance, and of sets of invocations, such as mutual exclusion and ordering constraints.

An abstract data type's functional specifications are obtained by presenting a model of the type, expressed as a set of conceptual data objects, and constructing assertions that define the input constraint and input/output relations for each operation defined for the type. An abstract data type's behavioral specifications are obtained by constructing assertions that characterize state sequences over the model that are associated with invocations of the type's operations.

A machine's functional specifications are obtained by constructing assertions that characterize the relationship between its output sequences and its input and visible state sequences. A machine's behavioral specifications are obtained by constructing assertions that characterize input, state, and output sequences that satisfy temporal order constraints like mutual exclusion and liveness and safety properties, and temporal metric properties like time performance and time-out.

The results of a design step are procedure, data type and machine designs, and requirements on lower level mechanisms which together imply that the design meets its specifications. The design process then returns to requirements analysis and design of the lower level mechanisms. Ideally, design proceeds in a net top-down fashion. By net we mean that a design step will be finished before the designs of the lower level mechanisms it uses are finished. This allows for controlled depth-first exploration and backup when infeasibilities are discovered. A new level is reached when the procedures and types which realize primitives of the upper level are to be designed and proved correct with respect to their requirements.

Machine, procedure, and data type designs are arrived at by procedural refinement (introducing subprocedures), type refinement (designing an abstract type model's implementing structure and its operations' algorithms), and object space partitioning (partitioning a machine's permanent objects into disjoint subsets, each of which is manipulated by a submachine).

For example, we recommend these four steps, in this order, for designing an abstract data type:

- Prepare a data type definition that specifies type's externally visible behavior and externally accessible operations.
- Design an implementation structure; express the mapping between the type definition's model and the implementation structure.
- Re-express the type definition's specifications in terms of the implementing structure, thus preparing the specifications to drive the design of type operations. The mapping function precisely defines this transformation.
- Design a data type refiner containing the implementation structure and procedures to implement the type operations. These procedure descriptions will satisfy the specifications expressed in terms of the implementing structure.

CSDL provides the specifications, mapping functions and data type models to carry out these steps. The process is similar for constructing machines:

- design external interfaces -- public objects,
- specify externally visible behavior in terms of the visible objects,
- introduce private objects, including other machines, to realize that externally visible behavior.

1.3.2 Object Orientation

The constructive approach guides designers in obtaining correct data types, procedures and machines by building down. CSDL's second major design guideline, object orientation, guides designers in building up a system from correct components. A system is viewed as a collection of data objects and procedural objects. System construction is the process of combining previously designed data and procedural objects to meet the system's goals, as expressed in the system's specifications. Object orientation cuts complexity because, during system construction, designers deal only with an object's external interface, as presented by its abstract model and operations (in the case of an abstract data type), or its public objects and external behavior (in the case of a machine). The same specifications that drive an object's constructive design explicate that object's properties and behavior for the purposes of system construction.

1.3.3 Complexity Management

Constructive correctness and object orientation combine for a design style in which complexity is controlled both in the design of individual system components and in the overall system design, and correctness is maintained during object design and system design by using specifications as goals. Complexity management occurs at three levels:

o within a single component:

Every system component -- module, data type, and procedure -- has a specification; the design which meets the specification is created separately.

o between a component and its clients:

A component has a public part: its specification and public objects (in the case of a module) or allowable operations and attributes (in the case of an abstract data type). A component's private part realizes its public specification. Modules containing objects which are instances of abstract data types see only the object's type specification, its allowable operations, and the attributes they can examine or manipulate. A type's representing structure and the procedures which implement its operations are private. Modules interacting with a module see its specification, the public objects that constitute its visible state, and the public objects through which they can exchange information with it. The module's procedures and internal data objects are private.

o among components:

CSDL allows the design and implementation of data types, procedures and module to be carried out independently of designing the system that uses all these objects.

1.3.4 Linguistic Support for the Design Guidelines

The table below matches detailed technical methods for constructively correct, object oriented design with the CSDL features that support each one.

TECHNICAL METHOD	LANGUAGE SUPPORT
1. Design mechanisms to match applications	Data Abstraction Facility Machine "Type" Facility
2. Use data objects of designer specified types	Data Abstraction
3. Object oriented design	Machine as unit of modularization Data Abstraction
4. Refinement	Subprocedures Data Type Refiners Machine Realizations
5. Constructive Correctness	Mapping Function for Data Types Weakest Precondition Semantics

Linguistic Support for Technical Methods

1. CSDL supplies simple primitives for defining datatypes and communication mechanisms. These primitives do not presume any particular mechanisms; rather, they give the designer the freedom to specify and subsequently design mechanisms that are most appropriate for the problem at hand.

2. CSDL encourages designers to augment built in types with high level, application oriented types, so that an application system can be designed in terms of the most meaningful objects for the application.

3. Object orientation conceives of a system as a collection of objects, each of which performs some task, cooperating to achieve the system's goals.CSDL has two encapsulating devices -- machines and data abstraction.

4. Refinement - adding design detail in a rational way - is supported with three techniques:

   o subprocedures to refine algorithms,

   o data type refiners to implement data type specifications, and

   o machine realizations to implement machine specifications.

5. Finally, a constructive methodology for creating designs that meet specifications rather than testing and adjusting designs <u>until</u> they meet specifications rationalizes the creative process.

1.4 CONSTRUCTS AND NOTATION

Language is the concrete tool we give designers. Any mechanism, such as dynamic process creation, or object, such as the communication port, that the language cannot express will not be in designs. Any design principle, like information hiding, that the language does not allow or makes difficult will not be used. So our goals were that CSDL's notation:

- meet the requirement of supporting technical personnel in creating, reasoning about, and presenting design specifications and descriptions in terms of the implementing technology,

- be the vehicle for carrying out a constructive design process that produces an implementation blueprint,

- facilitate unambiguous communication and permit verification through rigorous semantics based on formal models.

We also strove to meet general language design goals: readability, writeability, intuitive appeal for a user community which is comfortable with programming languages, and succinctness without sacrificing clarity. This section presents a notation which meets the three technical goals and is an honest but imperfect attempt to meet the human-engineering goals.

Section 1.4.1 briefly explains the structure of a system definition and mentions the CSDL constructs for expressing each definition element. Section 1.4.2 presents the description language for expressing sequential procedures, and the specification constructs that are applicable to sequential procedures. Section 1.4.3 presents CSDL's built-in data types and its facilities for defining abstract data types. The section deals with both ordinary passive data and with the active data types that constitute communication objects. Section 1.4.4 discusses machines, the building blocks for concurrent systems, and the language constructs needed to specify interactions among autonomous control sites. Section 1.4.5 presents CSDL's documentation format.

1.4.1 System Definition Structure

A system's definition is the union of its description (what components it contains) and its specification (how its components behave). Specifications (assertions) and descriptions (declarations and algorithms) are interspersed in each component's definition. A system definition is said to be correct if there are proofs (in some sense) that the system description satisfies the specifications.

CSDL's notation comprises descriptive constructs for stating declarations and algorithms and specification constructs -- atemporal assertions in the first order predicate calculus and temporal assertions in a variant of the Moszkowski-Manna [MOSZ83] temporal logic for specifying hardware behavior.

1.4.2 Procedures

Procedures, functions, and type operations are described in an algorithmic language based on Dijkstra's [DIJK76] guarded commands and specified by atemporal assertions that characterize relations among data.

1.4.2.1 Algorithmic Language

The algorithmic constructs are:

MEANING	NOTATION
no_operation	SKIP
sequence	<statement> ; <statement>
assignment	<id list> := <expression list>
procedure invocation	<id> (<parameters>)
type operation invocation	<qualified reference> (<parameters>)
non-blocking selection	IF B1-->S1 ▯...▯ Bn-->Sn FI
blocking selection	WHEN B1-->S1 ▯...▯ Bn-->Sn END
non-blocking repetition	DO B1-->S1 ▯...▯Bn -->Sn OD
blocking repetition	WHENEVER B1-->S1 ▯...▯ Bn-->Sn END

The formal semantics of these constructs (i.e., what happens when one of them is used) are given by a semantic function called the weakest precondition predicate transformer. These semantics are presented in [FRAN83a, Chapter 6]. Informally:

> No-operation, sequence and assignment have the usual meaning. Procedure and type operation invocation suspend the caller and transfer control to the invoked procedure. Procedures may instantiate objects; upon completion, a procedure's temporary objects disappear.
>
> Selection and repetition are nondeterministic guarded commands. Non-blocking selection and repetition have the semantics presented in [DIJK76].
>
> Blocking selection and repetition can test the same conditions as the non-blocking varieties and also test whether communication events (data arrival or departure) have occurred. Their guards may refer to active or passive objects; at least one guard should refer to an active object.

WHEN blocks until some guard(s) is (are) true, then executes the statement associated with a true guard. Since it is not required that any of the guards eventually becomes true, the statement may wait forever.

WHENEVER blocks until some guard(s) is (are) true, executes the statement associated with a true guard, returns to waiting until some guard(s) is (are) again true, and so forth. Once it is entered, there is no exit from this statement.

A procedure description consists of some optional declarations of local objects and an algorithm described using these constructs.

1.4.2.2 Atemporal Specification

The context for atemporal specifications is an external view of procedures as functional relations between inputs and outputs. In that context, it is useful to specify facts about state, such as preconditions, and facts about behavior such as the precondition/postcondition pair, which express a procedure's effect. These facts are specified through values of data objects and changes in those values.

CSDL's atemporal language is first order predicate calculus with extensions such as a facility for introducing local definitions and convenience features like a case construct for abbreviating a conjunction of implications. The language is described in detail in [FRAN83b, Chapters 3-8]. We divide atemporal assertions into "value propositions" and "transition propositions". Value propositions characterize state by asserting static relations among values of several objects ($x > y$) or between an object and its values ($x \leq 10$ OR $x > 10$). Transition propositions characterize state transitions by asserting a relation between a state and its (not necessarily immediate) successor ($X'=X + 1$).

CSDL uses two sorts of procedures, machine procedures and abstract data type type operations. Each sort's specification may contain the following elements:

```
Name ( <input parameters> ) RETURNS <type specification>
  PRE  <value proposition>
  POST <transition proposition>
  INVARIANT <value proposition>
  BEHAVIOR <assertion>
END
```

The optional RETURNS clause, which is part of the procedure description, is used for value returning procedures.

In machine procedure specifications, the proposition following PRE is a precondition which specifies the permissible machine states when invoking this procedure. In a type operation specification, PRE expresses constraints on the parameters and the instance; correct operation is guaranteed if the precondition is satisfied. The precondition can define a required relation among the procedure's local objects, between the procedure's parameters and its local objects, or among local objects, parameters and global objects. For a type operation, these objects may be elements of the type's conceptual model rather than actual objects in its represeuting structure.

The proposition following POST is a postcondition which specifies the relationship between the state of the machine or type instance at termination and the parameters and state of the machine or instance at invocation. If there is a return value then the relationship between it and the parameters and state of the machine or instance at invocation is also specified.

The optional INVARIANT specifies a relationship among the parameters and global data of the procedure or type operation that is preserved by an execution. It must be guaranteed that, if the invariant is satisfied when the procedure or type operation is entered, then it will be satisfied upon exit.

The optional BEHAVIOR section allows the designer to express any useful information about the procedure's function or properties that is neither a precondition, a postcondition nor an invariant. An atemporal assertion can, for example, express a resource constraint. A procedure's performance specifications expressed as temporal assertions (see Section 1.4.3.4) would also appear in its BEHAVIOR section.

In summary, a type operation or procedure's specification is a collection of atemporal assertions which, minimally, define a relationship between input and output states together with the constraints on the input. The intended interpretation is that when a procedure is invoked with the objects and parameters satisfying its input constraint, it is guaranteed to terminate with the objects in a state correctly related to the input state.

1.4.3 Data

Data objects hold all the information a system uses. Machine data are permanent; they last as long as their containing machine does, though their values may change over time (for example, a data base). Procedure data are transient; they come into existence when their procedure is instantiated and vanish when their procedure terminates.

Data objects are also either active or passive. A passive object undergoes a value change only when a procedure in its containing machine manipulates it. An active object may undergo a value change without its containing machine's operating on it. Intermachine communication occurs between active objects.

CSDL provides some built-in passive and active types, and a type definition facility.

1.4.3.1 Passive Data

An instance of a passive data type changes value only as the result of the invocation of some procedure or operation within the machine containing it. The assumption of passivity of objects lies at the heart of the basic theories for reasoning about a program by looking at its effects on data. In particular, the power of invariants is due in part to the assumption of passivity of the objects involved.

1.4.3.1.1 Built-in Passive Types

CSDL has four built-in scalar types: Boolean, Char, Integer and Real. Type Boolean has the usual value set (TRUE and FALSE) and operations (NOT, AND, OR, COR, XOR, CAND). Type Char has two operations, "assignment" and "equality test", and no pre-defined value set. Designers can define its value set to suit the intended implementation environment. Type Real denotes the mathematical reals. Type Integer denotes the integral reals, so every Integer data object is also a Real. CSDL provides the usual numeric, relational, and boolean operations; numeric, relational and boolean expressions are formed in the usual way. Initial value declarations are allowed for all scalar types. Using the abstract data type facility, fixed range Integer and Real subtypes can be defined.

CSDL provides four constructed types: enumerated types, records, discriminated unions and arrays.

An enumerated type is a finite set of elements; each element's only property is its name. There are both unordered and ordered enumerated types. All of the relational operators (<, >, ≤, ≥, =) are defined on elements of ordered enumerated types but only the relational operation equality (and, therefore, inequality) is defined on elements of unordered enumerated types. Assignment is defined on all enumerated types.

A record data object consists of a fixed finite number of data objects, called fields, which may be of different types. CSDL records are similar to records or structures in a number of Algol-like programming languages. Initial value declarations are allowed for entire records and for record fields.

Discriminated unions provide a facility for working with data objects that may contain values whose type is one of a finite set of types. They are similar to variant records in Pascal. A discriminated union's tag field is set automatically whenever its value field is changed. The tag field cannot be manipulated by any other means. The value field may be of any type.

CSDL provides Dijkstra arrays as the sequence abstraction. An array data object consists of a set of data objects, all of the same type, that is indexed by a contiguous range of integers; the set of objects may be empty. An array's size varies, shrinking when the object with the largest or smallest index is deleted and growing when an object with an index that is one more than the highest index or one less than the smallest index is added.

CSDL provides the array attributes and operations proposed in [DIJK76]. The special array attributes are:

(hib|lob) - an integer identifying the (largest|smallest) index of the array.

dom - an integer identifying the size of the array.

The special array operations are:

(high|low)_extend - a function which adds a new value to the (top|bottom) of the array, increases dom by one, and (increases hib|decreases lob) by one,

(high|low)_remove -a function which removes a value from the (top|bottom) of the array, decreases dom by one, and (decreases hib|increases lob) by one,

assignment - of a value to an arbitrary array element, or of values to an entire array with an array constructor.

access to arbitrary array elements - in the usual programming language manner.

Dijkstra arrays are more general than the usual programming language arrays, so they allow designers to describe more general information structures such as files, databases or dynamic memory.

1.4.3.1.2 Abstract Data Types

When a language allows designers to augment its built in types with high level, application oriented types, designers can work in terms of the most meaningful objects for the application. For example, in a compiler design, it is more meaningful to manipulate a symbol table object than to manipulate the more primitive objects that provide the symbol table. Furthermore, the benefits in complexity management of separating the use of a high level type from its definition are well documented in [LISK75, LISK79a, WULF76]. CSDL's abstract data type facility is a major design rationalization and complexity management feature.

A user-defined abstract data type is a set of values and a set of permissible operations on those values. User defined abstract types always have a definition which documents their externally visible behavior and externally accessible operations. If the designer decides there are no design issues involved in building instances of the type, no further design is required. If the designer decides there are design issues, then further design work is called for. The design of the type's representing data structure and operation implementations are packaged in a unit called a refiner, which contains the internal details that implement the external view and operations.

1.4.3.1.2.1 Type Definitions

Abstract types are defined using abstract model definitions, which are considered more understandable and easier for designers to construct than axiomatic definitions [LISK79b]. An abstract type definition consists of a model of the value set,(1) specifications of the allowable operations on the value set, and optional INITial and INVARIANT specifications.

The MODEL presented in every type definition is a device with which to express the specifications of the type's behavior. This conceptual model has nothing to do with how the type is eventually implemented. Its purpose is to give users of the type a picture of how the type behaves and what type operations accomplish. However, there is nothing to prevent a type's representing structure from being the same as its conceptual model.

CSDL has two kinds of type operations: ofuns which change the object's state and may return a value, and vfuns which return a value but do not change the object's state [ROBI77]. If a design undergoes formal verification, either in conjunction with its construction or after it is complete, only vfuns and not value returning ofuns may be used in expressions in the guards of IF, DO, WHEN and WHENEVER statements, because the semantics of these statements require that guard expression evaluation be side-effect free.

Type operation specifications were explained in Section 4.1.2. A type definition contains only the type operation specifications, which are presented in terms of the type's conceptual model. The type's refiner contains operation descriptions.

INITial states can be specified for an abstract data type's conceptual object space. An INIT assertion specifies a desired relationship among

(1) A model is a collection of typed objects, for example,

```
                    STACK (T:TYPE) IS
                    MODEL x: T ARRAY, tos:INTEGER
```

The MODEL objects' types indicate value sets for those objects; the operations defined on those types are not exported to the type under specification as permissible operations on that type.

the values of these objects. The intended interpretation is that, when a type instance comes into existence, its objects are guaranteed to satisfy the assertion.

An INVARIANT is a property of a type's MODEL established when an instance of the type is created. Each type operation must have the property that, if an instance satisfies the invariant and the operation is invoked so that its input constraint is satisfied, the instance will satisfy the invariant when the operation terminates. However, abstract type operation implementations may violate a type's invariant while they are in progress because the invariant is a guarantee to the type's users in their scope, not inside the type's scope.

CSDL also provides generic abstract types. A type definition may contain an optional parameter list consisting of pairs of the form <id list>:<type specification> or <id list>:TYPE. These parameters can be instantiated when an instance of the type is declared; they particularize a generic abstract type to an abstract type. The values of these parameters remain constant for the type instance's lifetime. <id list>:<type specification> specifies formal values. The <id>s in <id list> may appear anywhere that a value may appear, for example, in an assertion. <type specification> denotes a standard or user defined type. <id list>:TYPE specifies a list of formal names of known <type specification>s. Formal TYPE parameters may occur anywhere in the type definition that a type designator is required, for example, in the conceptual object space declarations and in operations' parameter lists or return clauses.

1.4.3.1.2.2 Type Refiners

A refiner is the package that contains the concrete decisions about how to represent an instance of an abstract data type and implement its operations. A refiner must contain:

- o the data structure chosen to represent an instance of the type,
- o one procedure for each operation defined on the type,
- o a mapping function that defines how the data structure corresponds to the model declared in the type definition.

A refiner is not a machine; it does not constitute a (potentially) asynchronous, independent locus of control. Rather, a refiner can be thought of as a set of templates of data structures and procedures. Instances of these templates replace uses of objects of the type being refined. Each declared object of the type may be thought of as being replaced by a distinct copy of the representing data structure, and each operation invocation may be thought of as being macro-replaced by an invocation of the refiner-procedure of the same name.

The operations specified in a type definition are implemented by procedures with the same names as these operations. This establishes the relationship between a definition's operations and the refiner's implementing procedures.

The relation between a type definition's conceptual object space and the refiner's representing structure is defined by a mapping function from the objects of the lower-level type refiner onto the conceptual object space of the data type to be represented. The mapping function is defined within the refiner because that is the only place where the lower level objects are visible.

The mapping function is used to uniformly substitute lower-level objects for upper-level objects in the type's specifications. The resulting assertions constitute specifications of initial states, data invariants, and procedure specifications which must be satisfied by the representing data structure and the procedures which implement the type operations in order for the refinement to be a consistent representation of the data abstraction.

1.4.3.2 Active Data

The idea and power of passivity of objects fits well with a single machine "running" in isolation. When a machine is put in association with other machines, and interacts with them, things become more complicated. One machine, A, can affect or interact with another machine B only by somehow changing the value of one of B's objects. From the point of view of machine B, some of its objects have "spontaneously" changed state; thus they are not passive. An active object is one that can exhibit a state change that is not the result of an operation or procedure invoked upon it within the machine that contains it. When machine A causes a spontaneous state change in B, there is a flow of information from A to B. The mechanism by which one CSDL machine can affect another involves a pair of complementary active data objects and their connection.

Every active type is an abstract data type whose model is composed of passive objects. It may be scalar or structured. Communication paths among CSDL machines are formed by connecting instances of complementary active types. Often the models for each of a pair of complementary types are identical, but each has a different set of operations. Some operations (for example, receive) absorb information into a machine's space; others, (for example, send) emit it into a machine's environment. The role of an active type (emitter or absorber) is determined by the type's operations, not its model. Structured types may even play the role of emitter and absorber. For example, each end of a full duplex channel could be specified as the same active type whose model consists of two data fields, with operations to emit through one field and absorb through the other. The channel is constructed by connecting one object's "emitter" to the other's "absorber," and vice versa.

The fact that every active type is an abstract data type allows the definition of various forms of blocking and non-blocking send and receive primitives, and ack/nack and time-out protocols.

1.4.3.2.1 Complementarity

The idea of complementary types is based on an intuitive "plug-and-socket" idea. Each member of a set of machines has a piece of communication equipment, an object of an active type. When the pieces are plugged into one another they form a plex over which the machines may interact with one another. The various pieces may be structured, so there may be several ways in which the components of a piece could be mated with the components of other pieces. Thus there must be a specification of the one desired way of mating the components.

We formalize these ideas in CSDL using the notion of active type MODEL compatibility. Two objects are compatible if they are of the same type. Two sets of objects are compatible if they can be put into a one-to-one correspondence so that the corresponding pairs of objects are compatible. Two sets are said to be complemented when a one-to-one correspondence between them has been specified. In the simplest case, complementary types may be built up by:

- defining an active type's MODEL as a set of passive objects,
- defining its complement's MODEL as a compatible set of objects, and
- complementing the two sets, that is, specifying the desired one-to-one correspondence.

More generally, a non-empty subset of one active type's MODEL objects is made compatible with a non-empty subset of another's MODEL objects, and these two subsets are complemented. This allows an active type's MODEL to contain objects that are available for specification purposes but do not participate in connections with other active objects.

Complementary types should be designed in tandem; the design process will produce a pair whose coupled effect is the communications protocol the designer is aiming for. However, each of the complementary types will be documented separately; each type's specification will contain a COMPLEMENTS specification that expresses the complementary relationship between the elements of that type's MODEL and the elements of its complement's MODEL.

For example, an active type T could be modeled as having two components, x and y, of type A, and one component, z, of type B. Another type U could be modeled as having two components, p and r, of type A and one component, s, of type B. T and U would have appropriate, different, operations. Types T and U are compatible because there exist one-to-one correspondences between them in which corresponding pairs are compatible.

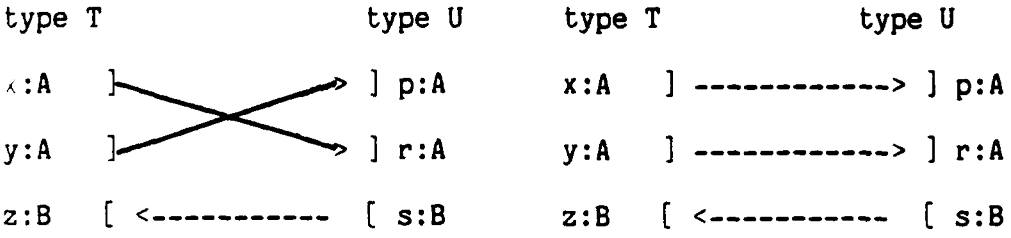
{x<-->p, y<-->r, z<-->s} and {x<-->r, y<-->p, z<-->s} are two such correspondences. T and U are complements once one of these correspondences is specified.

1.4.3.2.2 Connection

An isolated member of a complementary set of active objects is useless; in fact, members behave passively in isolation. However, once they are connected the resulting plex can exhibit active behavior; information can flow from object to object, and hence from machine to machine. The COMPLEMENTS specifications of the complementary types provide the semantics of connection. Those semantics are that each component of one object is associated with its complement in the connected object. The nature of the association is that the components have the same value throughout the period of the connection.

It is now clear how active behavior is obtained in a connected set of complementary active objects -- the invocation of an operation that changes a component of one object immediately changes the state of the components paired to it in other objects; the machines containing these other objects see spontaneous state change.

For the example in 4.2.2.1, an object of type T could be connected to an object of type U in either of the configurations shown below.



Complementarity of Types T and U

1.4.3.2.3 Inlets and Outlets

CSDL has two predefined active types. One, the outlet, allows a machine to send information to its environment; the other, the inlet, allows a machine to receive information from its environment [BOEB78]. These types are structured; their conceptual model is a record with two fields: a window that holds information of some type, and a Boolean flag. Inlet and outlet flags make transmission and communication detectable. Without flags, detection by comparing old and new window values fails when identical values are transmitted on the i-th and i+1-th transmission.

Each type has two allowable operations which may be performed only by procedures in a machine containing the object of the type. The inlet has a "get" for reading information and a "came" for testing its flag. The outlet has a "put" for writing information into it, and a "went" for testing its flag. These operations are invoked only by procedures in the machine containing the active object.

An inlet is an object from which information may be extracted by a get operation. A datum arriving from outside the inlet's containing machine will set its window to the arriving value and its flag so that inlet.came=TRUE. The first get performed after the arrival of a datum results in a new value being obtained; all gets after the first and before the next data arrival return the same value as the first get. This first get also resets the flag. Get, came, and data arrival are indivisible actions; a flag change cannot overlap the invocation of an operation. This rule constitutes the only guaranteed synchrony constraint.

An outlet is an object into which information may be deposited by an invocation of a put operation. Some time after invoking "put," an outlet's flag will spontaneously change so that outlet.went=TRUE if inlet.get is applied to its connected inlet. Only the last value put before a change to true will be communicated. Put, went and the flag change are indivisible, so the flag change cannot overlap the invocation of an operation; this constitutes the only guaranteed synchrony constraint.

The communication model based on connected inlets and outlets distinguishes between transmission and communication. Transmission between an outlet and a matching inlet is instantaneous; transmission is putting a value in an outlet. Putting sets the outlet's flag, the matching inlet's flag, and puts the value into the outlet's and matching inlet's windows. Communication happens when a value is got by a receiving machine. Getting absorbs a value into a machine's local space, and resets the flag on both the inlet from which the value is got and the matching outlet. Getting does not change the value in either object's window.

1.4.4 Machines

Concurrent systems are designed as collections of concurrently active, asynchronously communicating modules called machines in CSDL. These machines are instances of machine types.

A machine is a collection of data objects and a sequential procedure that manipulates those objects. The sequential procedure, Controller, may invoke subprocedures. A machine may contain objects that are themselves machines; in that case, the submachines operate concurrently with each other and with their parent, and each manipulates a disjoint subset of the parent machine's data objects. Each machine accomplishes a task. When a machine contains submachines, that task is accomplished by the parent machine and the collection of submachines.

Like abstract data types, machine types have a definition and a realization. A machine definition documents a machine type's externally visible data objects and behavior. A machine realization gives the internal details that implement that external view. The machine type is more limited than data types since there are no explicit operations defined on objects of type machine. Machine instances are created and destroyed in controlled ways.

1.4.4.1 Machine Definitions

A machine definition consists of a list of the machine's public objects and specifications of the machine's externally visible behavior.

Public objects are those (active and passive) machine objects which define the external view of the machine. A machine's realization is guaranteed to have these objects. A machine communicates with its environment through its active public objects. Its passive public objects are visible to the environment, but cannot be manipulated by it. Public objects are used in specifications of the machine's externally visible behavior.

Machine specifications may specify initial values, invariant properties and machine behavior.

An INITIAL assertion specifies allowable initial values of machine objects for every machine instance of the type; an implementation must guarantee that an instance will satisfy the assertion when it is created. An INVARIANT assertion specifies a property of the machine's objects which is satisfied when the instance is created and which is preserved at each state transition the machine undergoes. Procedure boundaries inside a machine are transparent with respect to a machine invariant; each statement in every procedure preserves the invariant. The invariant may be violated inside a type operation, but type operations are atomic from a machine's point of view, so the invariant is still preserved from the machine's point of view. BEHAVIOR assertions specify requirements and constraints on the machine's function and performance. These are atemporal and temporal assertions. Temporal assertions are explained in section 4.3.3.

1.4.4.2 Machine Realizations

A machine realization opens the black box machine definition. It is a package containing the concrete decisions about how to implement a machine's observable behavior. A realization must contain:

o the machine's public objects,

o the machine's controller, a distinguished procedure which is never invcked but starts executing when the machine is created. One typical controller structure is a prologue block which initializes the machine objects to some required state, followed by a loop for repeated scanning of inlets. In this loop, data arrival is responded to by invoking other procedures and sending data out through selected outlets. If this controller ever terminates, there can be no further response to data sent to it from other machines; also passive objects in the machine can no longer change state.

A realization may also contain:

- private data types and objects,

- machine and machine pool objects,

- specifications about internally visible behavior and performance, and about the relations between public and internally visible objects,

- subprocedures.

1.4.4.3 Dynamism

CSDL is intended for designing systems with inherent concurrency (for example, geographically distributed systems), systems in which concurrency is needed to deliver adequate performance, or for which expressing the design as a collection of concurrent modules leads to a simpler, more understandable design.

There are two basic concurrent architectures: the static architecture in which the system is created with a fixed number of modules which persist throughout its lifetime, and the dynamic architecture in which modules are created as needed to handle new tasks. CSDL supports them both.

In CSDL, the basic locus of control is the machine. The machine is a container of objects and a control procedure in execution. A machine may contain data objects of any type. A machine may also contain machine-objects, that is, other machines in operation, and pools of machine-objects from which operating machines may be created and destroyed. These structures (the machine-object and the pool of machine-objects) enable a single machine to contain several concurrently operating local loci of control.

1.4.4.3.1 Machine Creation

A CSDL system is a machine; a concurrent system is a machine which contains other machines. A system's, that is, a top level machine's, initial architecture comprises a collection of machines, each declared as an object in the object space of the root machine, SYSTEM. Each of these machines may contain machines, and so forth. When the system is instantiated, all machines in SYSTEM's object space are instantiated, communication links are forged, and they are set in operation. If these machines contain machine declarations, the same scenario is repeated. This is static machine creation. Machines created in this way cannot be destroyed explicitly; they cease to exist if the machine containing their declaration is destroyed.

When a machine containing pool declarations is instantiated, empty pools are created. During the machine's lifetime, its procedures may explicitly create and destroy pool elements. This is dynamic machine creation. Explicitly created machines are linked to their containing context as specified in an argument of the create operation. If the machine containing the pool objects ceases to exist, its explicitly created child machines cease to exist because the pools that hold them no longer exists.

Machines created statically or dynamically are wholly contained within their creating (parent) machine.

1.4.4.3.2 The Need for Pools

Pool structures are variable size collections of objects of some single machine type. (CSDL allows pools of machines only.) The collections are indexed by pool-unique names.

There are two operations on pools: "create," which adds an object of the machine type to the pool, and "destroy," which removes the object named by its index from the pool. It is also possible to select, or refer to, a particular element of the pool, and to ascertain the size of the pool, that is the number of operating machines currently in the pool.

CSDL provides dynamic machine creation and destruction to meet the real world requirement for dynamic process creation and destruction. CSDL puts dynamically created machines in pools to meet its goals of facilitating reasoning about designs and design verification. A pool's size attribute permits the specification of resource constraints ("This pool contains no more than 20 machines"), reasoning about pool size during design, and verification that a description satisfies specified bounds on resources. Of course, a design can contain pools whose bounds are not specified.

1.4.4.3.3 The Role of Public Objects

Every public object is part of a machine's externally visible state, but public objects serve different roles in a machine design. Public objects may, in addition to showing the external machine state:

- realize partitioning, when an object in the parent machine's object space is manipulated by a child machine in order to accomplish part of the parent's specified task;

- serve as communication ports, if they are active objects and if they are linked to complementary objects in their environment.

Public objects which only show visible state are never linked with the environment. Public objects which realize partitions are bound to objects in their environment. Public objects which serve as communication ports are connected to objects in their environment.

A machine's public objects are not intrinsically bindable only, connectable only or unlinkable. The same public object may be bound in one instance of a machine type, connected in a second, and unlinked in a third. The disposition of each public object is determined at machine creation by the initializing "linking specification" that appears in a machine object declaration or as a parameter to a create operation. Public objects that are not mentioned are unlinked at machine creation; they may remain unlinked for the machine's lifetime or may later be connected (but not bound) to complementary public objects in some newly created machine.

1.4.4.3.4 Communication

Dynamic system restructuring is complete only when a newly created machine is tied in to the rest of the system. This section discusses mechanisms for accomplishing that linking and presents the information flow issues involved.

Because CSDL is intended for designing operating system type applications, it must be able to express a range of communication options from paired, blocking send/reply through third party reply to non-blocking send and receive. One language facility that gives CSDL the flexibility to express many communication mechanisms is that it has two linking modes: binding and connection.(1)

A machine influences its environment by means of public objects that are linked to the environment by binding or connection. Unlinked public objects cannot influence the environment. A machine whose public objects are all unlinked is effectless.

(1) An equally important factor in accomodating a range of communication options is that CSDL's communication objects and primitives do not support a particular communication mechanism as, for example, Argus [LISK81] supports send/reply and CSP [HOAR78] supports rendezvous. CSDL's communication objects are inlets and outlets which can serve as communication ports and from which more complex abstract active objects can be built. CSDL's plug and socket communication model is neutral with respect to the kinds of communication and synchronization mechanisms the application under design contains. The application's designer builds the required commmunication protocols and synchronization mechanisms from these simple, neutral facilities.

1.4.4.3.4.1 Linking

Communication links may be forged only between newly created machines and machines that already exist. There are two varities of linking:

1. Binding - in which a parent machine identifies one of its actual objects with a child machine's public object, giving up access to that object for the duration of the child's lifetime. Both active and passive objects may be bound; binding occurs between objects of the same type. Binding is similar to parameter passing by reference, with the child's public object acting as the formal parameter and the parent's object as the actual parameter. After binding, an object which belonged to the parent belongs to its child. Only one object exists in the system; control over it shifts from parent to child and it is a semantic error for a parent to access or modify a bound object during its child's lifetime. That actual object's value is unchanged. Hence, the result of binding is that the child's public object is initialized to be the value of the object to which it is bound.

   There can be only one binding between a child's public object and a parent's object for the created machine's entire lifetime. Bindings are broken only when a machine is destroyed.

2. Connection - in which public objects in child machines are "actualized" and the parent machine declares information flow paths among them or between some of them and its own objects. Only active objects are connected, and connection occurs between complementary objects: an inlet is connected to an outlet and vice versa. Unlike binding, two objects are needed to forge a connection.

All binding is done at machine creation. Connection takes place only in the context of creation. Connection is done between a newly created machine and an already existing sibling or between a newly created machine and its parent. Hence, whenever a connection is made between two machines, at least one of them is in the process of being created; it is impossible to connect two machines if both of them already exist.

1.4.4.3.4.2 Information Flow When Forging Communication Links

Connecting a complementary set of active objects establishes information flow paths among their containing machines. The semantics of connecting complements is that their complementary parts effectively merge into one, so that parts of like types will have the same values for the lifetime of the connection. Since this identity is established at the moment the connection is made, there will be a one-time flow of information (usually garbage) into some of the connected machines as their objects undergo apparently spontaneous state changes. To avoid injecting garbage values into the state space of a machine in operation, we say that the active object in the machine being created is "assigned" the value of the complementary object in the machine being connected to it. Since every communication link is forged between a newly created machine and an already existing one, this convention insures that values do not flow into a machine already in operation. It is always possible to identify which member of an information flow path belongs to a newly created machine. This semantic for connection insures that we can reason about a machine's behavior and properties in isolation.

A machine may have an INIT or INVARIANT assertion which must be satisfied upon machine creation. When connection forging injects values into a newly created machine, those values must satisfy the machine's INIT and/or INVARIANT assertion.

The problem of garbage information does not arise in binding because control over the same object is transferred from parent to child and this transfer does not change the object's value. But information flow does occur, since the child's public object and the parent's object become one. The parent must guarantee that its object, when bound to a child's public object, will satisfy the child's INIT and/or INVARIANT specification, if any. In practice, it is safest, either to bind to a public object which is not mentioned in an assertion, or bind several objects to a set of public objects that participate in an INIT relation.

1.4.4.4 Temporal Specifications

Temporal specifications are needed as soon as the notion of several machines operating concurrently is introduced. When two or more processes progress concurrently and interact, we must be able to say things about that progress and those interactions. Atemporal specifications of the functional relation between a machine's inputs and outputs are not sufficient to talk about computational progress in the face of interactions. We need to specify phenomena like termination, synchronization, and scheduling. Those phenomena can be specified only by pointing at changes in data configurations in the time dimension, in other words, by characterizing a system's computational history.

Temporal assertions may express ordering relations (A precedes B) with no metric time attached, timing relations (A precedes B by two units of time), metric properties of states and transitions (this transition takes three units of time), and properties of data objects at particular points in a system history (x=0 after this transition).

Like the atemporal language, CSDL's temporal language also specifies behavior in terms of relationships among values of data objects. The essential difference between atemporal and temporal specifications is that temporal specifications are concerned both with values and with the order in which values and value changes arise in the system history. The temporal language is based on a temporal logic; its semantics are defined in terms of a set-theoretic model of computation and a model of time. The model of computation is based on primitive notions of data value and data object. The model of time is based on the real line.

CSDL: CONCURRENT SYSTEM DEFINITION LANGUAGE

We are currently experimenting with a temporal specification language based on [HALP83, MOSZ83]. This is documented fully in [FRAN83b, Chapters 10-11].

Every temporal proposition is an assertion about histories. Temporal specifications assert facts about the whole system history by characterizing sub-se-quences of that history and the order in which those sub-se-quences occur. The basic sub-se-quences from which assertions are composed are named by propositions from the atemporal language.

Some temporal propositions, called state propositions, characterize sequences that begin with a particular state. That is, they characterize a system as being in a certain state. Others, called action propositions, characterize sequences that begin and end with states that stand in some specified relation. They characterize a system transition. Temporal specifications that specify temporal partial orderings on the state sequences in a system history are built with state propositions, action propositions, and composites formed using conventional logical connectives. They specify properties of the entire system history by specifying the history's structure, a partial order of the states in the history. Given a particular present, we may specify both the future and the past of a computation's history. A specification about the future is: "If a message is sent by module A, it is eventually received by module B." A specification about the past is: "If module B receives a message, it was sent either by module A or by module C." We may also specify properties of the entire history; one such is: "There is never more than one token on the communications bus." Several temporal assertions may specify different structures for the system history; a correct system design must realize all the desired structures.

CSDL uses six temporal operators, <I>, <T>, <A>, [I], [T], and [A]. In order to explain their semantics, we introduce the following sequence notation: Let (R) and (S) be temporal propositions characterizing sequences, and let s = s0, ..., sn be a sequence. Then, informally:

(R;S) is true of s if and only if there is at least one state si in s such that R is true of the subsequence s0, ..., si and S is true of the subsequence si, ...sn, $0 \leq i \leq n$. The semicolon is the basic structure operator; it allows the expression of sequences in terms of ordered sub-se-quences. Left and right parentheses delimit sequences specified by their structure.

<I> (S) (read: sometimes initially S) states that S is true of s if and only if S is true of some initial subsequence s0, ..., si of s. We can express <I> (S) as

$$<I> (S) = (S;TRUE)$$

where TRUE characterizes all non-empty sequences and is used to build "don't care" sub-se-quences.

<T> (S) (read: sometimes terminally S) states that S is true of s if and only if S is true of some terminal subsequence si, ..., sn of s. We can express <T> (S) as

<T> (S) = (TRUE;S)

<A> (S) (read: sometimes somewhere S) states that S is true of s if and only if S is true of some subsequence si, ..., sj of s. We can express <A> (S) as

<A> (S) = (TRUE;S;TRUE)

[I] (S) (read: always initially S) states that S is true of s if and only if S is true of all initial sub-se-quences s0, ..., si of s. We can express [I] (S) as

[I] (S) = NOT <I> (NOT S)

[T] (S) (read: always terminally S) states that S is true of s if and only if S is true of all terminal sub-se-quences si, ..., sn of s. We can express [T] (S) as

[T] (S) = NOT <T> (NOT S)

[A] (S) (read: always somewhere S) states that S is true of s if and only if S is true of all sub-se-quences si, ..., sj of s, $0 \le i,j \le n$. We can express [A] (S) as

[A] (S) = NOT <A> (NOT S)

[] is analogous, in the temporal context, to the universal quantifier ∀. [] is a universal temporal operator; it asserts that every terminal subsequence, initial subsequence, or subsequence of the sequence under discussion has some property. <> is analogous, in the temporal context, to the existential quantifier ∃. <> is an existential temporal operator that asserts that there is at least one terminal subsequence, initial subsequence, or subsequence of the sequence under discussion that has the property specified.

1.4.5 Documentation Format

A CSDL design document is simply a collection of all the type definitions, type refiners, machine definitions and machine refiners, arranged in any reasonable way. We recommend the following "loose-leaf-notebook" style of documentation format: machine definitions appear in a flat machine dictionary; realizations appear separately from definitions. Type definitions appear in a flat type dictionary; refiners appear separately from definitions. By flat, we mean there is no nesting that scopes names.

One of the machines in the machine dictionary and the companion realization dictionary must be the distinguished machine, SYSTEM.

A machine realization may contain type definitions for types used only in that machine. A type refiner might contain private type definitions but not nested refiners.

All machine definitions and most type definitions are visible system-wide. This means a type or machine may use any type or machine definition in the type or machine dictionary. Some type definitions may be contained within a type refiner or machine realization; these types are available only to their containing structures.

The design document may contain several of each dictionary, and within each dictionary every entry is a separate item from every other. There is also no importance to the order in which items appears in the loose-leaf notebook. So documentation standards in different organizations can be accommodated by putting pieces of the system design together according to each organization's documentation standards.

The loose-leaf style puts the right pieces of documentation in the right hands. For example, a machine's implementor will use the machine's definition to produce its realization but will use only the type definitions of the types that machine contains. A machine's client, on the other hand, will use only that machine's definition and the definitions of its public objects. Obviously there must be tool support for combining and recombining text fragments into proper configurations for different users.

From the project management standpoint, the loose-leaf notebook is produced a piece at a time, so there are clear, limited, and fairly autonomous tasks to be managed. A tool which manages the design text can also collect project management data about changes, number of accesses, versions, and so forth.

1.5 EXAMPLE

This is an example of a machine type, Manager, which accepts requests from its environment and returns responses. Figure 1-1 shows Manager's definition. It has two public objects. Information enters Manager from the environment through "in," an inlet of type Request. Information flows from Manager to the environment through "out," an outlet of type Response. We assume that Request and Response are defined.

Manager's public behavior is specified in terms of its visible objects. The first clause of the behavior specification asserts that any terminal subinterval of the system history that starts with the i-th arrives transition at "in" contains the i-th leaves transition at "out," where i may be any positive integer. In other words, the future of each request arrival contains the corresponding reply transmission. The second clause asserts that the i-th reply put to "out" is a proper response to the i-th request gotten from "in," for all positive i. Here the LET facility is used to introduce temporary logical variables, rep and req, that are used in specifying the desired properties.

Figure 1-2 shows an architecture that will implement Manager's behavior. It consists of the two visible objects, in and out, a machine of type Handler, three private objects and one procedure, Controller. Controller accepts data from the environment through in and enqueues it in holding. It also dispatches jobs to Handler when Handler signals that it is ready to accept a new job. Manager's three internal objects are a queue, an outlet for sending jobs to Handler and a Signal_in for accepting Handler's signals.

The queue is an abstract data type; Figure 1-3 shows its definition. The queue's type operation specifications are the usual kind of data characterization specifications. The INIT specification says that a object of type queue is empty at instantiation. The INVARIANT bounds the queue's potential length.

Figure 1-4 shows the queue type's refiner. Although the refiner design is not needed for designing Manager, we include it here to demonstrate the use of mapping functions. Mapping functions allow mechanical transformation of a type definition's specifications, which are stated in terms of the type's model, into specifications stated in terms of the type's implementing data structure. A type's implementation can be verified against these transformed specifications. The refiner's INIT specification, the first two clauses of the INVARIANT, and all the procedure specification are direct translations of assertion that appear in the type definition. The remaining six conjuncts of the data INVARIANT are needed once the choice of implementing data structures is made. These six clauses are invisible to users of the type; they concern only the representing data structure.

Signal_in's type definition is shown in Figure 1-5. Its complements specification defines an active type that is Signal_in's complement (see Section 1.4.3.2.1). Its INIT specification says that the initial value of an object of type Signal_in is FALSE. Its one operation, ready, returns the value of the signal (TRUE or FALSE) and leaves the signal FALSE. This type is designed in tandem with designing Manager's controller (Figure 1-7). In particular, ready is designed as a non-blocking operation because it is to be used inside a blocking repetition construct.

At the level of designing Manager's realization, we are interested only in Handler's definition, which is shown in Figure 1-6. Like Manager's definition, Handler's consists of some public object declarations and a behavior specification. The specification's first clause asserts a liveness property of the Handler, that one reply is eventually put out for each request that arrives. The second clause is an assertion about the past rather than the future. It says that, for any positive i, the initial subinterval of the system history which ends with the i-th request arrival contains a terminal subinterval that begins with the departure of the i-th signal. In other words, the i-th request must have the i-th signal in its past. The public objects are an inlet, an outlet, and a Signal_out, which is Signal_in's complement. At this level of refinement, we do not need to know even Signal_out's specification. This is an example of the extent of separation of concerns that CSDL allows. Signal_out will be specified in the context of designing Handler, and implemented when convenient.

Figure 1-7 shcws Manager's realization. Manager contains four local objects. The object server is an instance of a Handler machine. It constitutes an autonomous control site running concurrently with Manager. Server comes into existence when an instance of Manager does. The linking specification following ":=" indicates how server's public objects are linked to its environment. Linking is always done in the context of machine creation. Server's outlet, done, is bound to Manager's outlet, out. Binding outlet to outlet means that for server's lifetime it controls one of its parent's objects; this allows server to return replies directly to its parent's environment. Job and want are local active objects through which an instance of Manager communicates with Handler. Job is connected to server's inlet; want is connected to server's Signal_out. Holder is a Queue object that Manager uses to hold pending Requests.

Manager's public specifications are identical to the ones in the machine type definition. Its internal specification asserts that the requests submitted to the handler are just those that had been previously received from the environment.

Manager's controller is a (non-terminating) blocking repetition statement which waits on two active objects. When information arrives at the inlet, in, (that is, when in.came=TRUE), and the queue, holder, is not full, the Controller gets the arriving data from in and enqueues it. When want.ready=TRUE and the queue is not empty, the controller gets a job from the queue and passes it to server, the Handler instance, by putting it into the Request outlet, job, that is connected to server's Request inlet, next. This controller has such a compact design because most of the design work needed to attain this functionality was invested in designing the data types inlet, outlet, Queue, and Signal_in.

```
Manager IS
  PUBLIC  in: Request INLET;
         out: Reply OUTLET

  BEHAVIOR
    ∀ i: Posint(i)
        [T]( <i> arrives(in, in')
             => <T> <i> leaves(out, out') )
    AND
    ∀ i: Posint(i) [
        LET req, rep: Request(req) & <i> in.get' = req
                      & Reply(rep) & <i> out.put(rep)
            [ response(req, rep) ]
        ]

END {Manager}
```

Figure 1-1: Manager Machine Definition

```
Thing_queue(Thing:TYPE,n:INTEGER) IS
 MODEL Thing ARRAY

 LET tq:Thing_queue(n)
 INVARIANT 0 ≤ tq.dom AND tq.dom ≤ n
 INIT tq.dom = 0

 OFUN  empty
    PRE   TRUE
    POST  tq'.dom = 0

 OFUN  enqueue (t:Thing)
   PRE   tq.dom < n
   POST  tq'.hib = (tq.hib + 1) AND tq'.high = t

 OFUN  dequeue RETURNS Thing
   PRE   tq.dom > 0
   POST  tq'.lob = (tq.lob +1) AND dequeue' = tq.low

 VFUN  is_full RETURNS BOOLEAN
   PRE   TRUE
   POST  is_full' = true IFF tq.dom = n

 VFUN  is_empty RETURNS BOOLEAN
   PRE   TRUE
   POST is_empty' = true IFF tq.dom = 0

END {Thing_queue};
```

Figure 1-3: Queue Type Definition

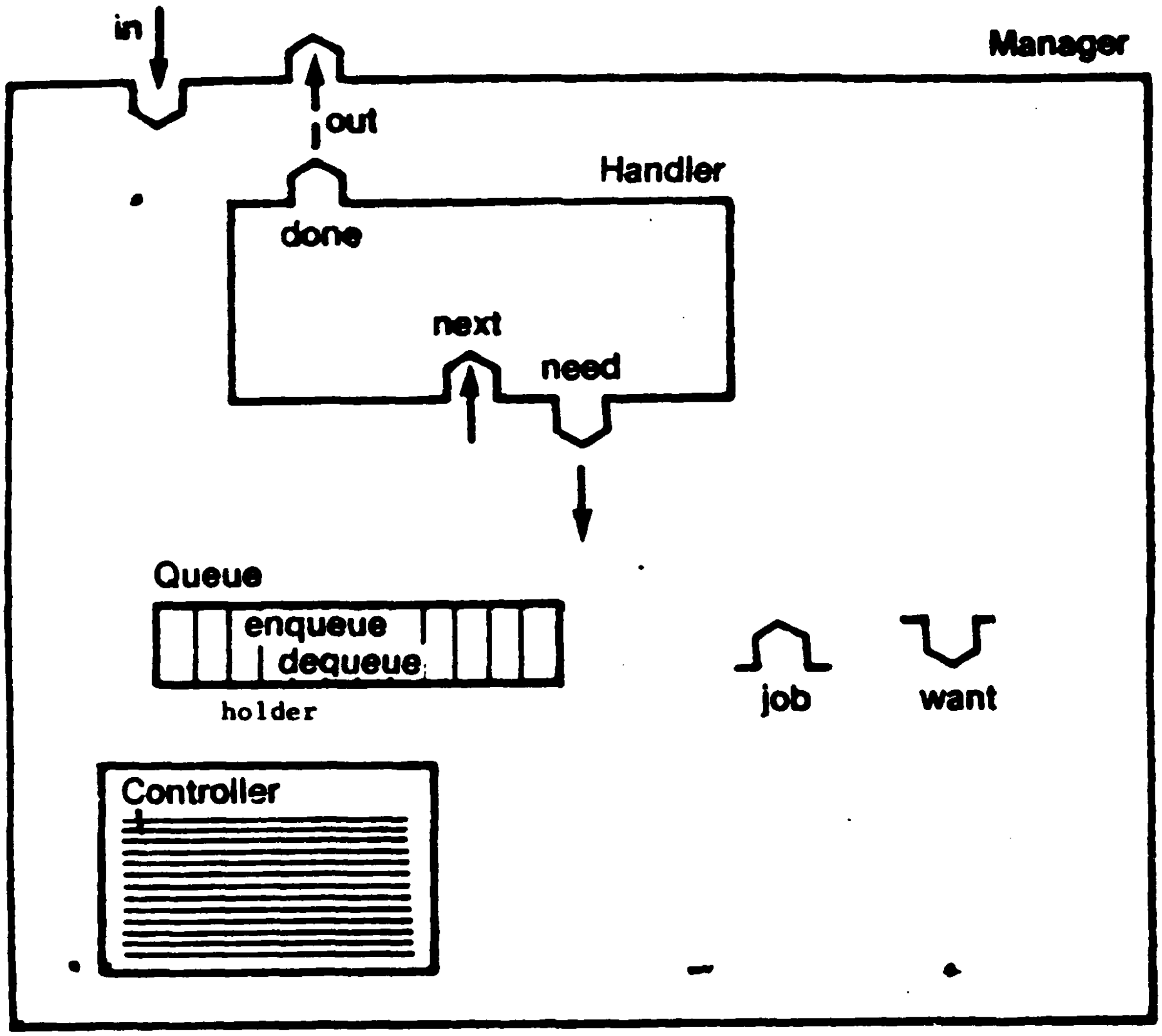


Figure 1-2 Manager Machine's Architecture

```
Q_as_buffer REFINES Thing_queue(Thing,n)

LET tq:Thing_queue

TYPES
  Circular_Buffer IS [circle: Thing ARRAY; front, rear:INTEGER]

OBJECTS
  buffer: Circular_Buffer

MAPPING

  buffer.rear REPRESENTS tq.lob;

  (buffer.front-1) MOD (n+1) REPRESENTS tq.hib;

  IF buffer.rear ≤ buffer.front
    THEN ∀ i:INTEGER(i)
        [IF buffer.rear ≤ i AND i ≤ buffer.front -1
         THEN buffer.circle(i) REPRESENTS
              tq((i-buffer.rear) MOD (n+1) + tq.lob)];

  IF buffer.rear > buffer.front
    THEN ∀ i:INTEGER(i)
        [IF[0 ≤ i AND i ≤ buffer.front]
            OR [buffer.rear ≤ i AND i ≤ n]
         THEN buffer.circle (i) REPRESENTS
              tq((i-buffer.rear) MOD (n+1) + tq.lob)];

  (buffer.front-buffer.rear) MOD (n+1) REPRESENTS tq.dom

INVARIANT
  0 ≤ (buffer.front-buffer.rear) MOD n+1 &
  (buffer.front-buffer.rear) MOD n+1 ≥ n &
  buffer.circle.lob = 0 &  buffer.circle.hib = n &
  buffer.front ≥ 0 &  buffer.front ≤ n &
  buffer.rear ≥ 0 &  buffer.rear ≤ n

INIT
  (buffer.front-buffer.rear) MOD n+1 = 0

{Procedures to implement type operations}

empty
  PRE  TRUE
  POST buffer'.rear = buffer'.front

enqueue (t:Thing)
  PRE  (buffer.front-buffer.rear) MOD (n+1) < n
  POST buffer'.front = (buffer.front + 1) MOD (n+1)
       AND  buffer'.circle (buffer'.front) = t
```

```
dequeue RETURNS Thing
  PRE (buffer.front-buffer.rear) MOD n+1 > 0
  POST buffer'.rear = buffer.rear + 1 MOD n+1 &
         dequeue' = buffer.circle(buffer.rear)

is_full RETURNS BOOLEAN
  PRE  TRUE
  POST is_full' = [(buffer.front-buffer.rear) MOD (n+1) = n]

is_empty RETURNS BOOLEAN
  PRE  TRUE
  POST is_empty' = [(buffer.front-buffer.rear) MOD (n+1) = 0]
```

Figure 1-4: Queue Type Refiner

```
Signal_in IS
 MODEL Boolean
 LET si:Signal_in; so:Signal_out
 COMPLEMENTS si,so
 INIT si=FALSE

 OFUN ready RETURNS Boolean
   PRE TRUE
   POST si'=FALSE & ready = si

  END {Signal_in}
```

Figure 1-5: Signal_in Data Type Definition

```
Handler IS
  PUBLIC  next: Request INLET;
          done: Reply OUTLET;
          need:Signal_out

  BEHAVIOR
    ∀ i: Posint(i)
           [T]( <i> arrives(next, next')
                => <T> <i> leaves(done, done') )
    AND
    ∀ i: Posint(i)
           [I]( <i> arrives(next, next')
                => <T> <i> signals(need, need') )

END {Handler}
```

Figure 1-6: Handler Machine Definition

```
Manager
  PUBLIC  in: Request INLET;
         out: Reply OUTLET

  BEHAVIOR {externally visible}
    ∀ i: Posint(i)
        [T]( <i> arrives(in, in')
             => <T> <i> leaves(out, out') )
    AND
    ∀ i: Posint(i) [
           LET req, rep: Request(req) & <i> in.get' = req
                         & Reply(rep) & <i> out.put(rep)
               [ response(req, rep) ]
           ]

  OBJECTS
    server: Handler := ( done::=out; {binding OUTLET to OUTLET}
                         next TO job; {connecting INLET to OUTLET}
                         need TO want {connecting Signal_out to Signal_in}
                       );
    job: Request OUTLET;
    want:Signal_in;
    holder:Request Queue

  BEHAVIOR {internal}
    ∀ i: posint(i) [
          [I]( <i> job.put => <T> <i> arrives(in, in') )
    AND
    <i> in.get' = THE req: Request(req) & <i> job.put(req)
        ]

  CONTROLLER
    WHENEVER ~holder.is_full & in.came -> holder.enqueue(in.get)
      [] want.ready & ~holder.is_empty -> job.put(holder.dequeue)
      END

END {Manager realization}
```

Figure 1-7: Manager Machine Realization

1.6 DISCUSSION

The software engineering project's goal is software engineering technology: formal models, design techniques, technical methods and languages. Because formal models are an engineering prerequisite for languages and technical methods, they have received much attention and are the most mature. Because language makes methodology and technical methods concrete, CSDL's language components are also relatively mature. In the context of CSDL we have done almost no work on design analysis techniques and tools, but the existence of models and formal languages means that the framework is prepared. Formality also means that the foundations for support tools are in place.

So far, the weakest precondition predicate transformer technique has been used rigorously by very few designers. Practitioners have been extremely reluctant to give up their familiar informal design styles for a formal design method that requires a large learning investment in basics like predicate calculus, in the CSDL notation and in the method itself. As long as we do not offer a tool that generates weakest preconditions from postconditions and algorithmic statements, we do not expect algorithms to be rigorously constructed in CSDL. However, the exercise of writing a specification informally using the constructive technique and examining an algorithm to convince oneself that it meets the specification does increase confidence in the design produced.

In reality, CSDL, with its "formal purity," is an investment in the future. When the need for probably or constructively correct software becomes so great that a large dollar investment in tools is warranted, CSDL will be available as a language whose constructs have proof rules and semantics defined in terms of a formal model. In the short term, parts of CSDL can be used along with less formal notations, for example, English language specification and designs produced in CSDL notation. This produces better designs than those created with a less complete notation and paves the way for designers to move into an entirely formal system. There is little specification support for properties like performance and reliability. Formalizations of these properties that can be related to the computational model are required in order to develop the linguistic mechanisms.

Appendix II

PUBLISHED PAPERS

(1) "An Object-Oriented Design Model for Reliable Distributed Systems" in the Third Symposium on Reliability in Distributed Software and Database Systems, Clearwater Beach, Florida, October 1983.

(2) "Zeus: An Object-Oriented Distributed Operating System for Reliable Applications" in the ACM National Conference, San Fransisco, October 1984.

(3) "Some Performance Models of Distributed Systems" CMG XV International Conference on Computer Performance Evaluation, San Fransisco, December 1984.

AN OBJECT-ORIENTED DESIGN MODEL FOR RELIABLE DISTRIBUTED SYSTEMS

Anand R. Tripathi
Pong-sheng Wang

Corporate Computer Sciences Center
Honeywell Inc.
Bloomington, MN 55420

Abstract

This paper describes an object-oriented design model for structuring reliable distributed systems. A system is viewed as a collection of objects that are accessed and modified by transactions. Recovery techniques are incorporated to make transactions atomic in the presence of component crashes and concurrent operations. Atomicity of transactions is based on constructing recoverable objects using multiple versions and commit protocols. These concepts are extended to nested transactions. The operations on distributed objects are peformed as remote procedure calls. This requires implementation of remote procedure calls in a reliable fashion. The facilities of reliable nested transactions and remote procedure calls are used to synthesize distributed objects that are highly reliable.

1.0 Introduction

The architectural features of distributed systems, such as physical isolation between system components which tends to reduce correlation among component failures, and redundancy of resources to support continued operations in the presence of component losses, offer great potential for designing reliable systems. This potential has remained largely unexploited, however, because of the lack of a formal discipline to integrate the known existing recovery techniques into distributed systems designs. In this paper we present a design model for distributed systems which facilitates a systematic and well-structured integration of known recovery techniques into the designs of distributed systems.

In constructing reliable systems, the maintenance of recoverable consistent states of objects is an important problem for system recovery. Another problem, which is functionally orthogonal to recovery, is concurrency control in distributed systems. The solutions to these two design problems interact closely.

Object-oriented designs offer an attractive approach to constructing reliable systems by confining errors in the system, by defining consistent system state to support rollback and restart, and by limiting propagation of rollback activities in concurrent systems. An object-oriented approach is comprised of objects accessed or updated by users through transactions, a sequence of primitive operations on a set of objects. A transaction is viewed as a unit of error recovery and synchronization in the system. The key to designing reliable systems is the atomicity of transactions and a sufficient level of redundancy in the system to support continued operations in case of loss of objects (i.e. system components).

Lampson and Sturgis [LAMP76], and Gray [GRAY79] introduced independently the concept of commit protocols to implement atomic actions on distributed objects in the presence of system crashes. The nested transaction facility is used for performing distributed concurrent operations. Constructing nested transactions introduces the concept of "conditional commitment". The commitment of a nested transaction is dependent on the commitment of the parent transactions. Most discussions on this topic have benefited from the concept of "sphere of control," first introduced by Davies [DAVI73]. A process execution is viewed as a "sphere of control" within which the process changes the states of the objects and controls the commitment of these changes. Once committed, the changes made within a "sphere of control" can never be revoked. The problem related to nested transactions, and the designs to address these problems have been discussed by Shrivastava [SHRI82b], Reed [REED78] and Moss [MOSS81].

In the proposed model, operations on remote objects are performed as remote procedure calls, requiring reliable implementation. Discussions on reliable remote procedure call models have appeared in recent literature, most notable the discussion by Spector [SPEC82], Nelson [NELS81], Shrivastava [SHRI82a] and Lampson [LAMP81b].

Creating small protected domains that interact through well-defined interfaces plays an important role in system recovery by confining error propagation. Object-oriented designs facilitate a systematic construction of such small protected domains for error recovery. Recently, the object-oriented designs have been used by Liskov [LISK82b], Shrivastava [SHRI81], Svobodova [SVOB81], Reed [REED79] and several others to structure distributed systems for high reliability. The scheme proposed by Reed was the first to use multiple-version facilities to implement atomic actions. This scheme has been used to implement a

This work was supported by RADC Contract No. F30602-82-C-0154

reliable storage facilities for objects in the SWALLOW [SVOB81] design. Liskov [LISK82b] has proposed object-oriented linguistic mechanisms based on the concepts of guardians and actions to construct reliable distributed systems.

The problems related to rollback of processes in concurrent systems have been addressed by several researchers [RUSS80] [KIM79]. One significant problem is the domino effect arising from unstructured interactions among activities and causing a cascade of rollback among concurrent activities. In object-oriented systems, the activities are transaction-based so that the interactions among activities are well-structured and disciplined. Again, "sphere of control" helps limit the rollback activity.

Managing redundancy in the system in the form of replication of objects or creation of backup objects is important for supporting continued operations in the event of loss of resources. The major problem in redundancy management is maintaining consistency among replicated objects, and having current state information with backup modules to support reconfiguration. The solutions to this problem keep a majority or a survivable set of the replicated copies in a consistent state.

In Section 2 we present the abstract design model for reliable distributed systems. Section 3 reviews briefly various recovery techniques applicable to distributed systems. Section 4 integrates these techniques into the design model. At each level of abstraction, appropriate recovery techniques are described.

2.0 *Design Model for Reliable Distributed Systems*

A design model, inspired by Lampson's lattice model [LAMP81a] for constructing reliable distributed systems, is shown in Figure 1. The objective of reliable distributed system designs is to synthesize secure and stable distributed objects that survive system crashes and support high function availability of services. Such objects are constructed using unreliable resources such as physical storage (disks), physical processor, and the communciation medium.

In this section we describe the design model shown in Figure 1 in a "bottom-up" fashion. This model is one possible approach to designing reliable distributed systems. It is particularly suited for an object-oriented systems in which interactions among objects are transaction-based. We identify the functions of each level in the graph shown in Figure 1. In Section 4 we describe the application of various recovery mechanisms to achieve these functions at each level of the design model.

The *physical storage* refers to non-volatile disk storage with a non-zero probability of information loss; for example, a page on a disk may be corrupted by a head-crash or other malfunction. Such failures can be characterized by reliability measures such as the mean-time-to-failure or a reliability function. Another problem with physical storage is the non-atomicity of write operations on pages, for example, a crash may occur in the disk system during writing a new value on a page, leaving the page corrupted because the old value has been destroyed and the new value has not been written completely.

The *stable storage* facility, constructed from the physical disk storage, provides atomic write operations on pages while enhancing the availability of data by replication.

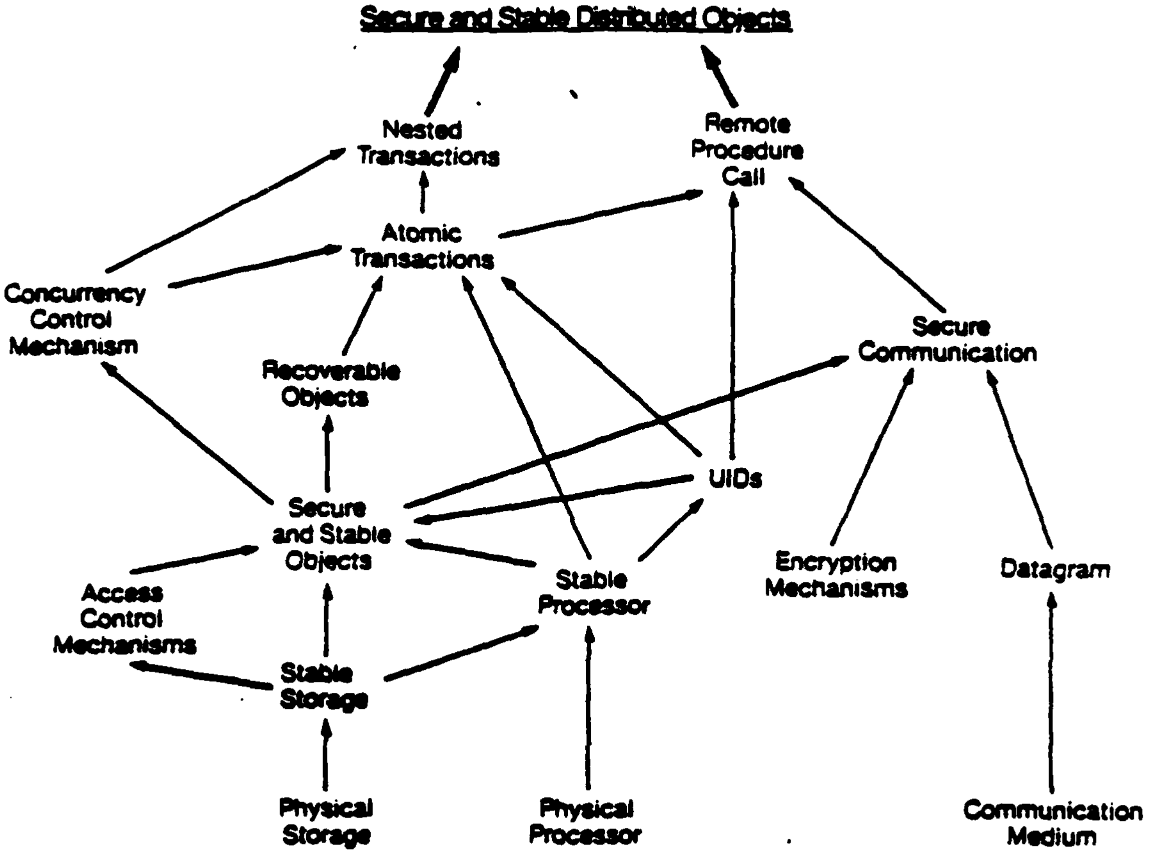
A *physical processor* loses its control state data on crashes. A restart operation causes a process to execute from the beginning. A *stable processor* facility, on the other hand, supports saving process states on the stable storage and restarting a process from a previously-saved process state. Saving process states is called *checkpointing.*

In our model the system consists of a collection of objects, each of which is of a well-defined *type*, managed by an *object manager*. In addition to supporting operations associated with the type definition, the object manager for a type also creates objects of that type, or destroys some existing objects of that type. A system-wide object called *type manager* facilitates the introduction of new type definitions in the system. This approach is based on the principles followed in the design of Hydra [COHE75]. The type manager object in our model corresponds to the TYPE-TYPE object in Hydra.

The next level of abstraction provides *secure and stable objects* based on stable storage, stable processor and *unique identifier* (UID) facilities. Stable objects are those that survive system crashes with a high probability and for which the primitive operations (i.e. the operations supported by the type definition) are atomic. Secure objects are protected objects which can be accessed only by authorized users. In our model, processes are considered as objects supported by a stable processor facility.

Using the UID facility, every object in the system is given a globally unique name. This name is never reused in the entire life-time of the system. From this unique identifier, the type of the object can be inferred. The UID also identifies the node where the object was created. Objects in the system may migrate from one node to another. The UID facility defines the logical name space in the system. Operations on an object are invoked by specifying the UID of the object and the operation name. Because the unique identifier allows determination of the type of that object, the operation invocation on an object is directed to the appropriate object manager for that type. Because the operations on the remote and local objects are invoked in an identical fashion, we find the remote procedure call paradigm a convenient abstraction.

The UID generation is based on the stable storage and the stable processor facilities. The UID generation facility is based on a local clock



A Design Model for Reliable Distributed Systems
FIGURE 1

process or a sequence counter that uses the stable storage to survive system crashes and to ensure that the same UID is not regenerated on restart of a node after a crash. The UID for an object indicates the type of the object and the node where it was created. A scheme for generating UID in a reliable fashion is described in [SCHA83].

The abstraction of recoverable objects provides mechanisms to restoring the state of an object after having made some changes to it, or committing a change to the object state. The concept of commitment forbids any restoration to states before commitment. Commitment of a change to an object implies permanence of the changes made to the object since the last commit operation.

We use the concept of immutable versions to implement mutable recoverable objects. An immutable object is one that is never changed once it is created, i.e., every change to an object creates a new object. In our model every change to an object creates a new version of that object; this version is uniquely identifiable by using the UID of the object and the version number. These principles have been discussed in detail in [REED78] and [SVOB81].

Atomic transactions are implemented using the facilities described above and some concurrency control mechanisms. A transaction should be atomic in the presence of concurrent operations and system crashes. Atomicity of concurrent transactions requires suitable mechanisms for concurrency control. There are basically two distinct approaches to concurrency control: locking protocols [ESWA76] and time-stamp based schemes [BERN81]. Recoverable objects support schemes to achieve atomicity of transactions in the presence of system crashes. Transactions in our model are treated as objects of process type. As in the case of any other object in the system, a transaction is assigned a UID.

Nested transactions enable constructing higher levels of abstractions by composing a set of transactions into one larger transaction. Nested transactions are also useful for introducing parallelism within an atomic action. The commitment of computations by each of the nested transactions is dependent on the commitment of the parent transaction. Concurrency control mechanisms are required to synchronize nested transactions of the same or different parent transactions.

Remote procedure call mechanism is based on the atomic transaction facility to ensure the atomicity of operations in the presence of system crashes and other concurrent transactions. The remote procedure call mechanism uses unreliable datagram facility with high probability for successful message delivery. The discussions in [SHRI82a] and [LISK82a] support building reliable remote procedure calls using less sophisticated facilities such as a datagram. These end-to-end arguments [SALT81] point out the wasteful duplication of functions at different levels. Secure communication is achieved by encrypting messages and storing non-encrypted messages in protected buffers.

3.0 A Review of Recovery Techniques

Conceptually, there are three fundamental strategies incorporated in every reliable system design. These are error detection, damage assessment, and error recovery. The following parts of this section describe the most common techniques for error detection and recovery.

3.1 Error Detection

The reliability of a design depends on the stringency of the techniques for detecting erroneous states in the system that may lead to system failures.

Some general techniques for error detection [ANDE79] are described below.

(a) Replication Checks: In such schemes, an activity is replicated and the results are checked for consistency. An inconsistency indicates a possible error condition. Errors can be masked by majority voting as in Triple Modular Redundant systems.

(b) Reversal Checks: This is used to check what the input to the system should have been. The calculated input and the actual input are compared for consistency.

(c) Coding Checks: This is the most popular form of error detection. Redundant information in the form of checksum or parity is associated with objects to detect erroneous states.

(d) Acceptance Tests/Consistency Checks: At certain well-defined points in the execution, tests are applied to the objects to ensure that the state at that point conforms to certain specifications. Any inconsistencies imply an erroneous state. Consistency checks can also be applied to some mutilated data structures that are reconstructed on recovery.

(e) Interface Tests: These tests ensure that the interactions among system components meet certain acceptance criteria. Tests are applied to the parameters and the results of interface functions to limit propagation of errors from one component to another through the interfaces. Confining errors is strongly dependent on the stringency of the acceptance tests. In distributed systems, interfaces provide well-defined and controlled means for the propagation of exception conditions between modules. If the interface function execution encounters error conditions, an error condition is returned to the caller through the interface.

(f) Diagnostic Checks: Explicit tests are conducted on system components for which expected outputs for given test inputs are known. The components to be tested and the components conducting tests should be independent. As pointed out in [ANDE79], diagnostic checks are rarely used as a primary error detection mechanism; but used rather as a supplement to other detection mechanisms.

(g) Interval Timer/Time-Out Mechanisms: In centralized systems, the interval timer technique for error detection is based on the time-out concept. Before starting an activity, the program starts an interval timer set to certain delay. If the activity is completed before the timer counts down to zero, the counter is restarted; otherwise, on counting to zero, the interval timer interrupts the process indicating some possible error condition. In distributed systems, time-out techniques are also used to detect possible error conditions. A process invoking a remote operation waits for a specified time-out period to receive the response. If no response is received within this period, an exception condition is raised and appropriate forward error recovery is initiated.

3.2 Error Recovery

Depending on the way a consistent system state is regenerated, error recovery techniques are divided into two broad categories: backward error recovery and forward error recovery. In backward error recovery, a prior consistent state in the execution history is restored. Forward error recovery techniques, which are application-dependent, use the present error state to arrive at some consistent state.

The backward error recovery requires facilities for establishing recovery points which, after crashes, support reconstructing or restoring the state at the most recent recovery point prior to the crash. Some of the techniques used for backward recovery are described briefly below:

(a) Checkpointing: In this technique, the complete state of the process to be checkpointed is saved on a stable storage. In a process-oriented design, a checkpoint also saves on the stable storage the current state of all the objects bound to that process. In effect a checkpoint creates a backup version on a stable storage of the complete execution environment of the process that existed at the time of checkpointing.

(b) Careful Replacement: This technique avoids updating objects "in place". Updates are made to a "current" copy, and a "shadow" copy maintains the version before the updates. On commitment, the "shadow" copy is replaced by the "current" copy.

(c) Multiple Versions: In this technique, updates to objects are recorded in a new version that becomes current only on the commitment of the updates. In case the updates are to be undone (i.e. aborted), the new version, which is uncommitted, is discarded.

(d) Logs/Audit Trail: In this technique, actions performed on an object are recorded in a log or audit trail. The purpose of the logs is to support either *undo* of the logged action for state rollback or *redo* the logged action to ensure permanence of results produced by some committed transaction. Logs that contain the redo actions are called the *forward* logs: logs that record the undo actions are called the *backward* logs. The backward logs either record the inverse operations or the values of the object before the application of the logged action. During a recovery process, backward log is used by scanning it backwards for undoing actions in a last-in, first-out fashion. The following *write-ahead* rule is always followed to ensure recovery: i) force the undo log on the stable storage before updating an object in-place, ii) force the redo log on the stable storage before committing an update.

(e) Differential Files: In this technique, all updates to an object are recorded on a differential file [SEVE76]. The updates from the differential file are merged periodically into the main copy of the object and such updates are then deleted from the differential file. The differential file technique provides an inexpensive means of maintaining multiple versions of a large object. *Intentions lists* are a form of differential files or forward logs containing redo actions that record the new values of the objects and have the property of idempotency. The property of idempotency implies that repeated executions (some of which may be incomplete) of this sequence of actions would always bring the updated object to the same state.

(f) Primary/Backup Mode of Operation: If an error is detected during the invocation of some service supported by the primary object, a backup object provides a continuation of these services starting with some previous consistent state. The backup object may not be identical to the primary object. The technique of *recovery blocks* [HORN74] is an example of integrating these concepts into software architectures. A primary block, along with one or more alternate blocks and an acceptance test, forms a recovery block. First the primary block is executed and the acceptance test is applied. If the acceptance succeeds, the recovery block terminates successfully; otherwise, the next alternate block is executed with the state of the system restored back to the one that existed before the application of the previous block.

(g) Object Replication: This technique maintains multiple copies of an object at different sites to increase its availability. At least a survivable subset is always kept in the most up-to-date state. This set is chosen such that the probability of all members of this set being in the crashed state is very low. Such sets are called the *atomic update sets* [MINO82]. The essence of this principle is reflected in some of the replication management schemes that have appeared in the literature. The simplest is the majority update rule [THOM79] proposed basically to address the concurrency control problem. A generalization of this scheme is the weighted-voting schemes proposed by Gifford [GIFF79] and Skeen [SKEE82], where every replica of an object is assigned some number of votes. The rules for accessing or updating the replicated object are based on acquiring sufficient votes (i.e. forming a quorum) in the system. All members in the quorum are updated atomically. By changing the rules for forming quorum for operations, different reliability and performance levels can be attained.

(h) Self-Identifying Object: In this technique suitable descriptors are attached to the objects to facilitate reconstruction of directories by salvation programs. Salvation programs are used only in cases of extreme failures where not enough information is left in a consistent state to support automatic rollback and restart. Such programs need operator intervention.

Generally, every reliable system design incorporates both forward and backward error recovery techniques. The most common technique for forward error recovery is *exception handling* [GOOD75]. Exception conditions are the anticipated error conditions in the system. An *exception handler* is a program block that is invoked when a specified exception condition arises during run-time. The purpose of the exception handlers is to bring the system to a consistent state. Generally the exception handlers are application specific. Forward error recovery requires a complete understanding of the application for which the system is being designed. In this paper we do not consider these techniques in any more details.

4.0 Integration of the Reliability Mechanisms in the Design Model

In this section we describe the reliability techniques that are suitable at each level of abstraction in the design model shown in Figure 1. The discussion is divided into four major parts: object management, transaction management, remote procedure calls, and the management of distributed objects. We focus on the problems related to recovery rather than protection and security issues.

4.1 Stable Storage

Lampson presented the techniques for constructing stable storage from unreliable disc storage facility [LAMP81a]. The primary goal of his scheme is to make the operation of writing disc pages atomic.

Lampson's scheme is based on the technique of careful replacement. The atomic operation for writing pages on the non volatile storage is called StablePut. The StablePut operation first writes the page on an unused disc page rather than writing it over the original; thus, any failure during execution of the StablePut operation leaves the original page intact. Periodically the two pages are compared, and the old page is replaced by the new one. The pages are also checked for any corruption of data by applying suitable parity or checksum tests. The corrupted page is replaced by the data of the other page if that page is still unspoiled. This replication of pages also increases the availability and mean-time-to-failure for the pages, provided the pages are stored on different storage units such that their failure is independent.

4.2 Object Management

An object manager, supports primitive operations on the objects of its type, as well as other functions such as the construction of recoverable objects, concurrency control, and access control. Objects for which recovery and synchronization are provided by the object manager are called atomic objects [LISK82b].

Generation of UIDs is an important part of reliable object management. A crash resistant scheme for generating UIDs in the system is described in [SCHA83]. In this scheme, every node in a subset of nodes, which forms a survivable set, must possess a stable storage facility. A global sequence counter is replicated over this subset and sometimes global synchronization among these nodes is required. On restart, nodes not having a stable storage obtain the sequence number from one of the members of this survivable set of nodes.

Recoverable Objects - Conceptually, constructing recoverable objects in our design model is based on the multiple version techniques [REED78], [SVOB81]. Every change to an object creates a new version of that object; such versions are finalized upon committing the enclosing transaction. On transaction aborts, the tentative versions created by that transaction are discarded. The versions are forced onto the stable storage to make them recoverable under node crashes.

Reliability techniques most suitable for constructing recoverable objects include multiple versions, differential files, intention lists, audit trails/logs, and self-identifying objects. Generally, a combination of several of these techniques is used in constructing recoverable objects at a node.

It is less expensive maintaining multiple versions as a differential file rather than as copies of the original object. A differential file in which the sequence of changes is idempotent can be used as an intention list to ensure the permanence of results on the commitment of a transaction. Backward logs are used for restoring objects by undoing the actions recorded in the log. Whenever a new uncommitted state of an object is to be forced in-place on the stable storage from the volatile memory, it is essential that (in order to keep the object recoverable) the backward log be forced on the stable storage before forcing the uncommitted object in-place on the stable storage.

Self-identifying objects and consistency checks play an important role during restart after a crash in reconstructing objects, object headers and directories during the restart after a crash. For example, with multiple versions, differential files and logs, additional information such as the object UID, state of the versions (committed, uncommitted, commit pending, etc.), pointers to other versions, logs and differential files is incorporated for crash recovery. After reconstructing the data structures on crash recovery, the consistency checks are important in checking the validity and correctness of the reconstructed data structures.

At this point we describe a scheme and its associated data structures for maintaining multiple versions in the system to construct recoverable objects. Logically, every version in this scheme contains a descriptor which contains the UID of the object, version number, UID of the transaction currently holding this version, a time-stamp indicating its creation time, and a status of the version. The status field can be in any one of the following states: uncommitted, commit-pending, committed, and aborted.

The time-stamp field of the versions is useful for discarding the versions created by a transaction since its last checkpoint. The commit-pending state is used during execution of the two-phase commit protocol [LAMP76], [GRAY79] with the current user transaction. The commit protocol is initiated by the user transaction by sending a prepare-to-commit message to the object managers of all the objects it has updated. On receiving such a message, the object managers change the status field of the current versions to commit-pending, and return a positive acknowledgement. A version in the commit-pending state cannot be unilaterally discarded by its object manager.

In the scheme proposed here, we use differential files to maintain multiple versions. The versions of an object are maintained in a differential file as records of changes to the existing committed copy of that object. Applying these changes to the object has the property of idempotency; therefore, the differential files also serve as intention lists. For every transaction, one such file is created as shown in Figure 2. The file control block (FCB) plays an important role in this scheme. The FCB for a differential file has two parts: Current Transaction Descriptor and Physical Storage Map. Current Transaction Descriptor contains the identifier and the status of the transaction that has recorded new uncommitted versions of the object in the differential file. Physical Storage Map points to the records on the stable storage containing the updates for the new versions. By rewriting this FCB using the atomic StablePut operation the entire FCB can be changed in one atomic action. This use of FCB for atomic updates is similar to the scheme described in [LORI77] [PAXT79]. To record an action on the file, the changes are written on new pages, the FCB is modified and re-written using the StablePut operation. At this point, the change has been successfully recorded.

4.3 Process and Transaction Management

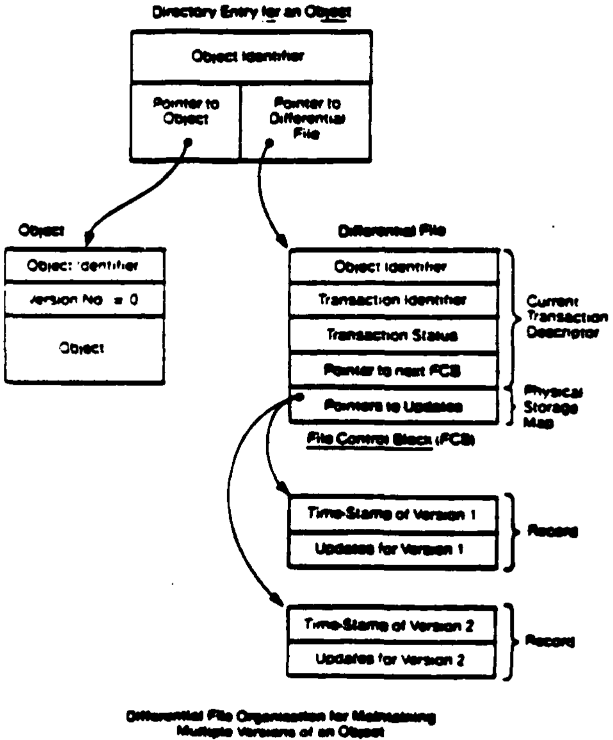
In this section, we discuss the use of reliability techniques to implement reliable processes and transactions. As noted in Section 3, processes are considered as objects. Transactions are atomic processes; transactions are objects of process type with some additional properties; therefore, transaction type is a sub-type of process type.

A process object once created can be in one of five states: Inactive, Running, Suspended, Completed, or Aborted. The operations for process objects include: Create, Destroy, Start, Restart, Status, Suspend, and Resume.

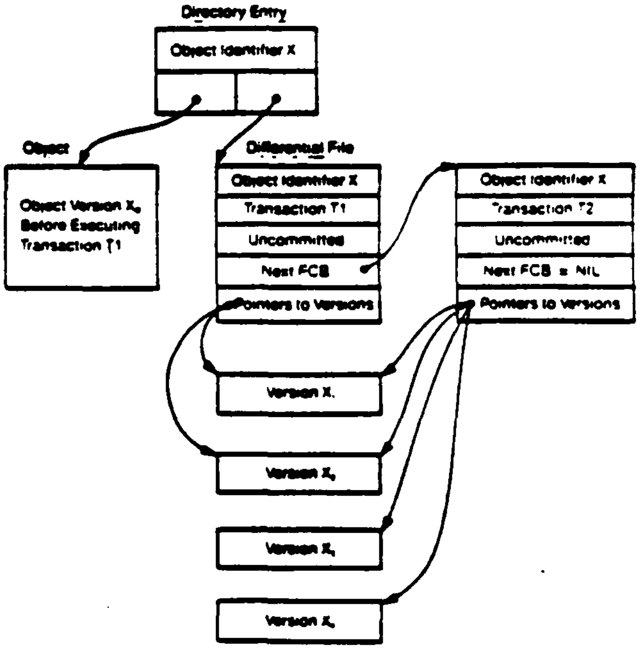
From an object-oriented viewpoint, new versions of a process object are created during execution of the process, i.e., a new version of the process object is created whenever its program counter changes. This view is consistent with the one for multiple versions of data objects. There is, however, a difference between these two types of objects in handling rollback recovery: with data objects, it is usually possible to save all versions of an object before these versions are committed so that rollback to a previous version is relatively simple; with process objects, it is too expensive and impractical to save the process states of all execution steps. Checkpointing can be viewed as the selective saving of versions of process objects and is used to establish recovery points for process objects.

The following operations for process objects are used to support checkpointing and rollback:

- o Establish_Recovery_Point - saves the current process state of the process object in stable storage.
- o Discard_Recovery_Point - discards the checkpoints of a process object.



Differential File Organization for Maintaining Multiple Versions of an Object
FIGURE 2



Multiple Versions of an Object for Nested Transactions
FIGURE 3

- o Rollback - continues the execution of a process from a checkpoint.

Note that with the above scheme for checkpointing, only the state of the process object is saved; the states of objects modified by that process are not saved in the checkpoint. This approach may create problems for error recovery since not all state changes of the process are recorded. It is necessary, therefore, to follow some discipline in using checkpoints and atomic transactions.

First, we require that a non-transaction process (e.g., a user process) must invoke a transaction in order to modify an object or a set of objects. The changes to an object are recorded as new versions of the object. New versions of the object are committed to become permanent at the end of a successful completion of the commit protocol among the invoked transaction, the invoking process, and the object managers of the modified objects. Uncommitted versions are discarded by explicit abort commands from the transaction process or by timeout on inactivity.

If a transaction is nonidempotent, i.e., multiple executions of the transaction produce different results, a problem may arise in error recovery since rollback of the process may cause a committed transaction to be re-executed. One solution to this problem is to always force the invoking process to perform a checkpoint before the transaction completes committing the modifie' objects. Checkpointing is part of the commit protocol; if the protocol determines to abort, the checkpoint is discarded. With this mandatory checkpoint, rollback recovery of a process can avoid undesirable repetition of transaction execution; however, this may cause too frequent checkpointing of the invoking process. The second solution, therefore, is to make checkpoint of the invoking process an option that is to be specified at the time of invoking a transaction. This checkpoint apparently is not required for idempotent transactions to guarantee correct execution; a process, however, invoking an idempotent transaction may elect to force a checkpoint during the transaction commit protocol for efficiency reasons. For example, if the transaction requires extensive computation compared to checkpointing the invoking process, and if the possibility of a failure is significant, it may be desirable to have a checkpoint as described above. That decision is left to the process that invokes the transaction.

The following example illustrates the flexibility provided by the second solution. Consider the following scenario in which a process receives some item from a buffer, then processes the item. GetItem is the transaction that is invoked by the process to receive an item from the buffer. Commitment of this transaction leaves the buffer in a new state in which the removed item is no longer present and the previous state can never be restored. If the process invoking this transaction checkpoints itself on the commitment of the transaction, the received item is a part of the checkpointed state of the process. Any subsequent rollback will restart processing of this saved item, and there will not be any need to re-invoke the GetItem transaction. On the other hand, if the process does not checkpoint on committing the GetItem transaction, on a subsequent rollback the old item would be lost; the process would invoke the GetItem transaction once again; and processing would be performed on a new item. In certain applications such as process control systems, the second scenario may be a valid mode of error recovery.

A question on checkpointing still exists: Because the checkpoint of a process does not include the current states of the objects that are modified by the process, how does it guarantee correct rollback recovery? This question can be answered by considering the ways in which objects are affected by a transaction: first, a transaction may directly modify an object by invoking an operation on the object; second, it may invoke another transaction that modifies the object.

We first consider the object modified by the transactions by invoking an operation on the

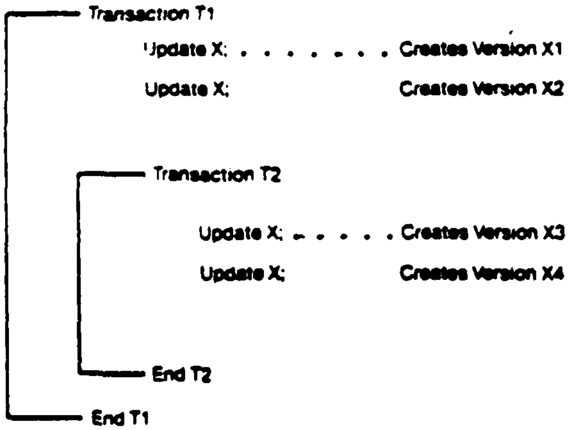

Transaction T1
Update X; . . . . Creates Version X1
Update X; Creates Version X2
Transaction T2
Update X; . . . . . . Creates Version X3
Update X; Creates Version X4
End T2
End T1

Figure 4

(a)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1		
(b)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1	Version 3 Uncommitted T2	Vesion 4 Uncommitted T2
(c)	Object X Committed None	Version 1 Uncommitted T1	Version 2 Uncommitted T1	Version 3 Uncommitted T1	Version 4 Uncommitted T1
(d)	Object X Committed None	Version 1 Committed None	Version 2 Committed None	Version 3 Committed None	Version 4 Committed None

Figure 5

object. One requirement for correctly implementing rollback is that the object manager for transactions must maintain for each transaction a list of UIDs of the objects that are affected by the transaction so that in the event of failure, the transaction will be rolled back to its latest checkpoint. This requires that all changes to objects made by the transaction after the checkpoint be discarded. The list of UIDs of the objects that are affected by the transaction provides a means for notifying these objects to discard the unwanted versions. This list is also used at the end of the transaction to conduct the commit protocol.

The discussion in the preceding paragraph implies that for implementing rollback, timestamps must be recorded with each checkpoint and each version of objects. This is necessary because checkpoints do not record all versions of a process object and thus there is not a one-to-one mapping between process checkpoints and versions of objects affected by the process. These techniques allow us to rollback a process correctly with modifications to objects in the first way.

The correctness of rolling back a process with the second way of modifications to objects is guaranteed by the principles followed in committing a nested transaction. The updates made by a nested transaction are made permanent only if its parent transaction is committed. Any rollback within a transaction may cause abortion of some committed nested transactions. In case a transaction is aborted, the changes made by the transaction are discarded (transaction are atomic). The objects are brought back to the state before the transaction was started. Failure of the invoking process poses no problems to these objects.

For idempotent transactions, the case is even simpler. Because transactions are atomic, changes to objects are either not done or made permanent from the invoking process point of view; and because the transactions are also idempotent, the rollback recovery is always correct, no matter where the invoking process is rolled back to.

Nested Transactions - As described in the previous section, at the end of a successful transaction the objects that were changed by the transaction are committed to become permanent; however, for a nested transaction, (a transaction invoked by another transaction) that completes successfully, commitment of the changes to objects will be dependent on the success of the parent transaction. If a nested transaction is aborted, the changes to objects made by the transaction will be discarded regardless of the success of its parent transaction; however, the failure of a nested transaction may not always cause its parent transaction to abort. In this section, we will describe a technique for implementing nested transactions. This technique requires only minor modifications to the technique for implementing single level transactions.

The technique that we use here requires recoverable objects as described in Section 4.2, i.e., each update to an object creates a new version of the object. Each version carries the information to indicate whether it is a committed, uncommitted, or commit-pending version. In order to support nested transactions, additional information is needed for each version to indicate on which transaction the version is dependent. This information is attached to a version when it is created by a transaction.

At the end of a transaction, the transaction either commits or aborts the changes to the object. If it aborts, only the uncommitted versions that are dependent on this transaction are discarded. If it commits, all versions of the object that are dependent on this transaction are changed to become dependent on the parent transaction of the current transaction. In this case, if the current transaction is at the top level, i.e., if it is invoked by a non-transaction process, these versions are committed to be permanent.

Figure 4 shows an example of nested transactions. Transaction T1 updates the object X to create Vesions 1 and 2. Both versions are uncommitted and contain the information that they are dependent on T1. A logical view of this result is shown in Figure 5(a). T1 then invokes transaction T2, which creates Versions 3 and 4 of X (Figure 5(b)). When T2 is completed successfully, all versions that are dependent on T2 are changed to be dependent on T1, the parent transaction of T2 (Figure 5(c)). Since T1 is a top-level transaction, i.e., it is invoked by a non-transaction process; when it is completed, all versions that are dependent on T1 are committed to become permanent (Figure 5(d)). Version 4 of X is now the current committed copy of object X; other versions can be discarded at this point.

In order to implement the above scheme for nested transactions, we can use differential files to maintain multiple versions as described in Section 4.2 and Figure 2. In the example in Figure 4, for a nested transaction, a new FCB and descriptor block is created as shown in Figure 3. When a nested transaction completes, the transaction UID field in the descriptor of the version that is being committed is replaced by the UID of its parent transaction, and the status field is changed to the uncommitted state. The status field of a version changes to committed only when the transaction committing it is the outermost level transaction. In Figure 3, transaction T2 is nested within transaction T1, and T1 created versions X1 and X2 for object X. Transaction T2 appends new versions X3 and X4 to the differential file, and these changes are visible only in the FCB that is being used for transaction T2. On the commitment of T2, the old FCB is replaced by the new one, and the user transaction field contains T1. When T1 commits, the status field in the descriptor is changed to committed, and the updates from the differential file are applied to the object. If any crash occurs during this updating, the procedure can be restarted from beginning.

4.4 Remote Procedure Calls

The problems related to reliable remote procedure calls have been discussed in [LAMP81b] and [SHRI82a]. One problem associated with the implementation of remote procedure calls is their execution semantics. In case of a retransmitted request message, should repeated executions be permitted? To address this problem, Nelson [NELS81] has classified the semantics of remote procedure calls as follows:

o "At most once" - In this semantic, at most one execution of the procedure takes place. It is possible that no invocation occurs. In this case the call returns with some error condition.

o "At least once" - This semantic means a successful return from the call guarantees at least one execution of the procedure.

In most of the applications "at most once" is preferred. One problem in the implementation of "at most once" is detecting duplicate requests at the server end. If the client process crashes after sending the call request and retransmits the request after the restart, the server should be able to detect the duplicate request. For this purpose the UID facility is used to assign a unique name to the request.

If a requester crashes after the server has started the procedure execution, the procedure invocation is termed an "orphan". After a restart from the crash, the requester process will retransmit the remote procedure call request. At this point we have two options in the design. The first option is to retransmit the request with the same UID as was used for the initial call request. If the original request was lost, this retransmitted request will invoke the remote procedure. If the server received the original request and started procedure execution that was later rendered "orphan" due to the requester crash, the server would detect the duplicate request, continue the "orphan" execution which is no more an orphan, and return the results of the "orphan" to the restarted requester. This scheme requires that every requester process must have access to a stable storage facility to store the request along with its UID so that on a restart the retransmitted request has the same UID. Because of this limitation we reject this scheme and propose the second scheme in which every remote procedure call is an atomic action which commits only after executing a commit protocol with the requester. Thus, the results produced by the "orphans" are discarded because the commitment protocal fails. This scheme eliminates the need for a stable storage at every node at the expense of decreased performance due to commitment protocols.

The reliability of the datagram facility can be enhanced by introducing appropriate reliability techniques into the network layer and the link layer supporting this facility. At the network level, the network topology is an important design issue. A network topology with higher connectivity would generally exhibit better reliability characteristics. At the link level, appropriate retransmission protocols are used to deal with transient errors.

4.5 Distributed Objects and Transactions

The reliability techniques at this level deal with maintaining redundancy in the system. The redundancy in the system is maintained in the form of object replication, primary/backup copies, or survivable sets of objects. The techniques suitable for managing redundancy at this level are based on the principles of voting [THOM79] [GIFF79] together with some commit protocol [LAMP76] [GRAY79]. At the distributed application level, the reliability techniques deal with the synthesis of reliable objects by redundancy management and the construction of recoverable transactions. Atomic transactions play a key role at this level. At the application level, these fundamental mechanisms are integrated into some higher level techniques, such as a recovery block, for system structuring. Forward error recovery based on exception handling is an important part of reliability techniques at this level.

The problems associated with the management of redundancy in the system have been discussed in Section 3. The nested transaction facility provides a convenient and powerful abstraction to perform atomic operations on a set of distributed objects. Replication management techniques based on quorums or majority consensus are used within nested transaction structures.

The concept of recovery blocks can be used conveniently at this level to define a primary transaction along with a set of backup transactions and some acceptance test. This can be done easily in our model because transactions are atomic. Integrating the backward recovery techniques, such as a recovery block, with forward error using exception handling can create very effective recovery mechanisms in a design. Such an integration of these two concepts has been described in [MELL77]. For forward error recovery, exception conditions can be associated with primitive operations on objects. Exception handlers can be introduced within a transaction; this does not affect the atomicity of a transaction. If a transaction is a part of a recovery block, an acceptance test is applied on its completion, but before its commitment. The transaction is committed only if this test is passed or else the transaction is aborted and an alternate transaction is tried.

Conclusions

We have presented an object-oriented design model that supports structuring of distributed systems for high reliability and error recovery. In this model, we have identified the error recovery problems at the different levels of functional abstraction and have shown how various error recovery techniques are integrated into this design model. For example, techniques based on multiple version concept are used for contructing recoverable objects, checkpointing and commitment techniques are used for constructing atomic transactions, and the techniques based on replication and primary-backup modes of operation are used for constructing reliable distributed objects. This model has been used in the design of Zeus [BROW83], an object-oriented distributed operating system for high integrity applications.

Acknowledgements: We would like to thank Professor James C. Browne and Vincent Fernandes for providing constructive and supportive comments on this work.

References

[ANDE79] Anderson, T., Lee, P.A., Shrivastava, S.K., "System Fault Tolerance," Computing Systems Reliability, Cambridge, England, 1979, pp.152-209.

[BERN81] Bernstein, P.A., Nathan Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2, June 1981, pp.185-222.

[BROW83] Browne, J.C., V. Fernandes, J.E. Dutton, A.R. Tripathi, P. Wang, "Zeus: An Object-Oriented High Integrity Distributed Operating System," Technical Report, Corporate Computer Sciences Center, Honeywell Inc., Bloomington, MN, 1983.

[DAVI73] Davies, C.T., "Recovery Semantics for a DB/DC System," ACM National Conference, 1973, pp.136-146.

[ESWA76] Eswaran, K.P., et. al., "The Notion of Consistency and Predicate Locks in Database Systems," Communications of the ACM, 19, 11, November 1976, pp.624-633.

[GIFF79] Gifford, D.K., "Weighted Voting for Replicated Data," Seventh Symposium on Operating Systems Principles, 1979, pp.150-162.

[GOOD75] Goodenough, J.B., "Exception Handling: Issues and Proposed Notation," Communications of the ACM 18, 12, December, 1975, pp.683-696.

[GRAY79] Gray, J.N., "Notes on Database Operating Systems," in Operating Systems: An Advanced Course, ed. R. Bayer, R.M. Graham, and G. Seegmuller, Springer-Verlag, 1979, pp.393-481.

[HORN74] Horning, J.J., et. al., "A Program Structure for Error Detection and Recovery," Computer Science, Springer Verlag Lecture Notes in "Operating Systems Techniques", Volume 16, .

[KIM79] Kim, K.H., "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," First International Conference on Distributed Processing, 1979, pp.284-295.

[LAMP76] Lampson, B.W., Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System," Technical Report, Xerox, Parc, April 1976.

[LAMP81a] Lampson, Butler W., "Atomic Transactions," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siegert, Springer-Verlag, 1981, pp.246-265.

[LAMP81b] Lampson, Butler W., "Remote Procedure Calls," in Lecture Notes in Computer Science Vol. 105, ed. B.W. Lampson, M. Paul, and H.J. Siegert, Springer-Verlag, 1981, pp.365-370.

[LISK82a] Liskov, B., "On Linguistic Support for Distributed Programs," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp.203-210.

[LISK82b] Liskov, B., Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Ninth Annual Symposium on Principles of Programming Languages, January 1982, pp.7-19.

[LORI77] Lorie, R.A., "Physical Integrity in a Large Segmented Database," ACM Transactions on Database Systems, March 1977, pp. 91-104.

[MELL77] Melliar-Smith, P.M., Randell, B., "Software Reliability: The Role of Programed Exception Handling," ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, March 1977, pp.95-100.

[MINO82] Minoura, T., Wiederhold, G., "Resilient Extended True-Copy Token Scheme for a Distributed Database System," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982.

[MOSS81] Moss, J. Elliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA 02139, April 1981.

[NELS81] Nelson, B.J., "Remote Procedure Call," Technical Report-Department of Computer Science, Carnegie-Mellon University, May 1981.

[PAXT79] Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," Seventh symposium on Operating System Principles,, December, 1979, pp.18-23.

[REED78] Reed, D.P., "Naming and Synchronization in Decentralized Computer Systems," Technical Report MIT/LCS/TR205, September 1978.

[RUSS80] Russell, D.L., "State Restoration in Systems of Communicating Processes," IEEE Transactions on Software Engineering, Vol SE-6, No.2, March 1980, pp.183-194.

[SALT81] Saltzer, J.H., Reed, D.P., Clark, D.D., "End-To-End Arguments in System Design," Second International Conference on Distributed Computing Systems, 1981, pp.509-512.

{SCHA83] Schantz, R., et. al., "Cronus:A Distributed Operating System - Interim Technical Report No.2," RADC Report No. 5261, February 1983.

[SEVE76] Severance, D.G., Lohman, G.M., "Differential Files: Their aplication to the Maintenance of Large Databases," ACM Transactions on Database Systems, Vol. 1, No. 3, September, 1976, pp.256-267.

[SHRI81] Shrivastava, S.K., "Structuring Distributed Systems for Recoverability and Crash Resistance," IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981, pp.436-447.

[SHRI82a] Shrivastava, S.K., Panzieri, F., "The Design of a Reliable Remote Procedure Call Mechanism," IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982, pp.692-697.

[SHRI82b] Shrivastava, S.K., "A Dependency, Commitment and Recovery Model for Atomic Actions," Second Symposium on Reliability in Distributed Software and Databae Systems, July 1982, pp.112-119.

[SKEE82] Skeen, D., "A Quorum-Based Commit Protocol," Proc. 6th Berkeley Workshop on Distributed Databases and Computer Networks, Berkeley, Calif., 1982, pp.69-80.

[SPEC82] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network," Communications of the ACM, Vol. 25, No. 4, April 1982, pp.246-259.

[SVOB81] Svobodova, L., "A Reliable Object-Oriented Data Repository for a Distributed Computer," Eigth Operating Systems Principles Conference, 1981, pp.47-58.

[THOM79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp.180-209.

This work was supported by Rome Air Development Center Contract No. F30602-82-C-0154.

Zeus: An Object-Oriented Distributed Operating System For Reliable Applications

James C. Browne*, James E. Dutton,
Vincent Fernandes, Annette Palmer
Information Research Associates**

Jonathan Silverman, Anand R. Tripathi,
Pong-sheng Wang***
Honeywell, Computer Sciences Center

1.0 INTRODUCTION

This paper presents the principles followed in designing Zeus, an object-oriented distributed operating system designed to study integration of recovery mechanisms into the designs of distributed command and control systems. The primary goal of the Zeus design is to define reliable object management functions for distributed command and control systems and to evaluate the performance and the correctness of the recovery mechanisms for these functions. Therefore, no implementation of this design currently exists. The user provided functions support definition of object types, creation of objects, and updating of distributed objects using atomic transactions. We are currently evaluating the performance characteristics of this design using simulation models and proving the correctness of the recovery mechanisms using formal methods based on Gypsy language [AKER83], events and state transition based models [TRIP83b], and simulation models. To achieve these goals we have refined the Zeus design to a significantly detailed level. To date we have explored this design only from the viewpoint of these goals. Several research problems necessary to implement this system remain unexplored. For example, a linguistic mechanism is needed to introduce object type definitions into the system and to define processes and transactions.

A distributed operating system for highly reliable applications must provide 1) recovery mechanisms that are transparent to the application developers and 2) naming mechanisms that make the physical distribution of objects and functions transparent to the application programmer. The second feature is important to make development of distributed software no more difficult than the development of conventional software systems. The Zeus design has made a significant contribution in this direction. Other systems have integrated these two concepts in their designs, however they typically limit object management to the file storage level. To date, Argus [LISK82] is the only other system which provides a set of general mechanisms for reliable management of distributed objects of any type. Zeus provides these mechanisms and addresses several other issues such as object relocation, authentication and object protection, not included in the Argus design. Another novel feature in Zeus is the integration of the conventional database management functions into the operating system object management functions. This is important because most of the todays popular operating systems do not provide efficient mechanisms for database applications [STON81]. Even with respect to its recovery model, the Zeus design differs significantly from other known designs.

Much of the recent research in reliable system design is actually exploration into system structuring techniques. These are more significant for distributed systems than conventional centralized systems because distributed systems are intrinsically more complex. A structured approach can reduce design complexity by factoring the designs into layers that create different levels of functional abstraction; the design of a layer can then be carried out somewhat independently of the design of other layers. The layers in the system can be viewed as creating horizontal partitions in the system design.

Another structuring concept, which is dual as well as orthogonal to layering, is object-orientation which creates vertical partitions in the system. The interactions between partitions occur through well-defined interfaces;

This work was supported by Rome Air Development Center Contract No. F30602-82-C-0154

* Names of authors are given in alphabetical order

** Work by Information Research Associates was performed under a Purchase Order from Honeywell, Inc.

*** This author is currently employed with IBM Santa Teresa Lab., San Jose, CA. This work was performed by the author while employed with Honeywell Inc.

thus, each partition in the system represents an independent domain where the internal structure of a domain can not be directly accessed by other domains. A vertical partition essentially embodies the concept of objects in the system. The whole system is viewed as a collection of objects. All state transformations in one partition by other partitions are performed through the interfaces defined by the partition. The advantage of such an approach is that the design of the internal structure of any given partition is independent of the designs of other partitions. These are the fundamental principles of data abstraction. From the viewpoint of reliable system design, such an approach is very attractive because it supports confinement of errors within an object boundary. This also implies that the recovery mechanisms for a given partition can be designed to suit its reliability requirements.

The concept of object-oriented design has been used in some recent distributed system designs such as Cronus [SCHA83], SWALLOW [SVOB81], Argus [LISK82], and in the approach presented in [SHRI81]. Argus provides object-oriented linguistic mechanisms for constructing reliable distributed systems, and SWALLOW provides reliable object managment. These systems do not support some of the other operating system functions such as access control, naming, sharing, and resource management. Some of the functions supported by Zeus, such as naming, authentication, and interprocess communication, exist in other operating systems such as Pilot [REDE80] and Grapevine [BIRR82], developed for network-based applications. Neither of these two systems are however, general purpose distributed operating systems.

The Cronus operating system design has significantly influenced the design of Zeus, largely because both these systems are intended for highly reliable applications such as command and control systems. Zeus provides users with reliable object management, which is not present in the current design of the Cronus system. Like Cronus, Zeus has the character of a general purpose operating system mainly because the nature of the command and control applications includes a wide range of processing characteristics. This is in sharp contrast to the requirements for banking or airline reservation systems where the application environment is well-defined. Zeus provides capabilities for defining and creating objects and transactions required by the application systems. It also provides mechanisms that support management of such objects in a reliable fashion. Zeus can be used for constructing any high reliability application system.

This paper presents the basic object-oriented building block mechanisms provided by the Zeus distributed operating system. The concept of object managers is the basis for system structuring. An object manager provides the encapsulation for a given type of objects; all objects of that type are accessed or updated via that object manager. The object-oriented recovery model underlying the Zeus design is described in [TRIP83a]. In this model the construction of reliable distributed objects is based on an atomic transaction facility and a remote procedure call mechanism. This approach is summarized in Figure 1.

DISTRIBUTED OBJECT MANAGMENT FUNCTIONS
(Partitioned and Replicated Objects)

REMOTE PROCEDURE CALL MECHANISM

ATOMIC TRANSACTION FACILITY

LOCAL OBJECT MANAGEMENT FUNCTIONS
(Concurrency Control, Recovery, Access Control,
Object Storage Management)

KERNEL FUNCTIONS
(Host Resource Management, Communication, Scheduling,
Interrupt Handling)

HARDWARE

A Model for Reliable Distributed systems
Figure 1

The lowest layer in this figure represents the kernel functions that execute at every host node of the distributed system. Above the kernel layer are the local object management functions such as storage management, access control, synchronization, and object recovery. This layer represents the functions that are associated with every object manager in the system; the functions at this level deal only with the centralized object management. The next layer provides facility of atomic transactions; thus, a sequence of operations can be performed on a set of objects in an atomic fashion. The remote procedure call mechanism facilitates operations on objects that are not local. We have adopted the remote procedure call mechanism because it provides a uniform way of accessing remote as well as local objects. Thus location of the object is transparent to the users during access or update operations. It is important to make the semantics of remote and local procedure calls identical in the presence of host crashes and communication link failures. In our design we have adopted the "at most once' execution semantics for remote procedure calls; thus, in the presence of duplicate messages or on server node crash-restart, effectively only one execution of the remote procedure will occur. The combination of the remote procedure call mechanism with the atomic transaction facility is used for managing objects that are either partitioned or replicated. Based on these mechanisms one can suitably create type definitions for replicated or partitioned objects such that one can access or update those objects in the same manner as updating centralized objects.

The object management model used in the Zeus design is based on the concepts developed in the Hydra [COHE75] design. In an object-oriented approach, the system is comprised of a set of objects, and each object is of a well-defined type. A Type Manager object for a given type manages all objects of that type. All operations on permanent and shared objects in the system are executed via their type managers. There are some obvious differences between the protection models used in the Hydra and Zeus designs. The protection mechanism in the Zeus design is based on access control lists while the Hydra model is capability based. Although both these models are equivalent in terms of their functionality, they differ with respect to their operational environment. The prime reason for using the access control list model in our design is to be able to change the access rights dynamically. Although it is not very efficient to change access rights dynamically in a capability based system, dynamic changing of access rights is important in a command control system where some of the nodes might be taken over by hostile forces.

2.0 PRINCIPLES OF DISTRIBUTED OBJECT-ORIENTED DESIGN IN ZEUS

2.1 Structure of Object-Oriented Systems

An object-oriented system consists of a collection of Type Managers and the objects created by them. As described above, the Type Managers create vertical partitions in the system. For a given type in the system, a Type Manager would exist at all those nodes which may be required to store objects of that type. A Type Manager at a node manages all objects of that type at that node. The multiple instances of Type Managers for a type function cooperatively to provide the abstraction of a single Type Manager for that type in the system. Each Type Manager defines an address space in which all the objects of that type reside. A Type Manager is logically viewed as a single process that performs all the state transformations on the objects in its address space in response to execution requests by some other objects of the same or different type.

At a physical node, several different Type Managers may reside, each managing objects of its type at that node. The abstract machine to support such an object-oriented system can be constructed from almost any hardware/software system architecture. The system architecture of the processors to support such a system must have: (i) a mechanism for switching the processor between Type Managers, (ii) a mechanism for partitioning secondary memory resources among Type Managers, and (iii) a mechanism for exchanging messages between Type Managers.

It can be seen from the preceding model of Type Managers that there is no concept of a system-wide state or uniform control and/or recovery mechanisms. Resource management functions and recovery mechanisms are partitioned along with the set of Type Managers. The traditional functions of system-wide software units such as operating systems and database systems are incorporated into a collection of Type Managers which implement the basic elements of the model of distributed computations. This is a radically new view of operating systems.

Object Type Managers are the primary building blocks for the permanent elements of the system. The Type-Type Manager is an object in the system that manages "types" in the system. It is the means by which new types are introduced into the system. The concept of the Type-Type Manager is essentially the same as that of the TYPE-TYPE object in the Hydra design [COHE75]

The objects in the system are accessed in a uniform fashion regardless of their locations. All operations on permanent objects are performed within a transaction. A transaction is bascially an atomic action that is defined as a sequence of operations on local or remote objects. A transaction ensures atomicity of distributed operations. It is possible to introduce concurrency within a transaction by creating nested transactions.

2.2 Object Naming

The most basic requirement at the lowest level of the system architecture is to identify and refer to objects unambiguously. This requires that each object must be associated with some system-wide unique identifier (UID). In the design model adopted for Zeus, a unique identifier is associated with every object in the system; from this identifier the "type" of the object can be inferred. To aid object location the Zeus design uses the concept of an "extended" UID. An extended UID adds a "host hint" field to a UID that identifies the host from which the object was most recently accessed. Based on the "type" field in the UID, a reference to an object is directed to the appropriate Type Manager at the node given by the "host hint" field of the UID.

2.3 Functions of the Type Managers

The functional characteristics implemented by the Type Managers are the original basis for defining abstract data types. Extending abstract data type concepts to include a formal basis for the integration of recovery, synchronization, and access control mechanisms generates a number of additional functions for the Type Managers:

1. Each Type Manager is directly responsible for the mapping of the occurences of the objects they define to physical storage.
2. Each Type Manager implements access control policies for the occurrences of its type.
3. Each Type Manager supports concurrent execution of its procedures and/or functions.
4. Each Type Manager ensures the consistency of the objects it stores under concurrent and distributed use.
5. Each Type Manager implements the necessary levels of redundancy to ensure the level of fault tolerance given in its specification.

This obviously integrates many functions that have been conventionally associated with database systems into the object management functions of this operating system.

2.4 Structure of Type Managers

A Type Manager is externally viewed as a collection of functions and procedures which can be invoked on the objects of its type by specifying the identifier of the object along with the operation name. This causes an invocation request message to be sent to the Type Manager regardless of its physical location in the system. Internally, these operations are executed by the Type Manager using one or more server processes; such server processes may be dynamically created or destroyed by a Type Manager. The operations on remote and local objects are invoked by the clients in the same fashion as procedure call. Such invocations on remote objects are performed by implementing remote procedure calls [NELS81] [SHRI82] with "at most once execution" semantics. A Type Manager consists of:

- Data structures for the objects of that type;
- Procedures/functions defining the type;
- Concurrency protocols;
- Recovery mechanisms;
- A database to manage the objects in its domain;
- A controller process that schedules/executes the requests.

A Type Manager is responsible for the permanent storage of the object instances of its type. Each Type Manager interfaces directly with some set of permanent storage devices. The Type Manager generates the mapping from the UID for an object of its type to the physical storage on some permanent storage devices. It also realizes object instantiation in the executable volatile storage from the permanent storage. There is no system-wide file system. The object management system takes the place of a file system.

A Type Manager consists of a controller process whose purpose is to schedule server processes to serve client requests. The server process is given the same UID as that of the client process. Thus, a client process is conceptually viewed as migrating into the address space of the Type Manager. This view of the migrating client process is useful from the viewpoint of enforcing access rights associated with the client process. On the completion of the requested service, the server process is deallocated. The controller process accepts the incoming or outgoing invocation request messages, performs security checks, and interfaces with the kernel procedures. Effectively, the controller process plays the part of a local operating system for the Type Manager; the scheduling policies can thus be tailored to the specific requirements of the Type Managers. The controller process manages the server processes performing the operations and provides them with a set of procedures that perform resource management, communication, protection and other services that are normally provided by an operating system.

A Type Manager's controller has several responsibilities related to protecting its objects from unauthorized access. Upon receiving an invocation request, the controller must obtain and store the requesting process' identification. This information is made available to the operation via a callable procedure so that the Type Manager's controller may check the access list of the object. In addition, the controller appends the identification of a process which is making an outgoing invocation request to some other Type Manager.

When an incoming invocation request is received, the controller attempts to locate the object whose UID is given in the request. First, the controller looks for the object in its own local pool of objects. If found, the program which will perform the operation on the object is parameterized with the object's local address and then is scheduled as the server process. If the object is not found locally, the controller determines if a "forwarding address" has been left for that object. This might occur if the object has been relocated to some other host. If the object is not found locally, the controller sends a reply message indicating that the object was not found and includes the forwarding address if any.

In response to an update request, the Type Manager creates a new version of the object. This version is committed only when the transaction that created it commits; the uncommitted versions are discarded if the transaction aborts.

Each Type Manager maintains a database which records the necessary information pertaining to the objects in its address space. This database records the identifiers of the objects of that type currently present at that node, their physical addresses, and the commitment status of their most current versions. A Type Manager is also responsible for aborting an uncommitted version if it detects no activity by the transaction that created this version. Every time a new version of an object is created by a transaction by invoking an update operation, the Type Manager ensures that this new version is written onto the stable storage before sending an acknowledgement for the operation. A scheme for maintaining such multiple versions using differential files is described in [TRIP83].

Type Managers are responsible for ensuring that each of their defined operations is atomic. The operation must either complete successfully or else abort, leaving the object completely unmodified. This is not difficult to achieve if only local objects are being modified in the operation. However, if the operation involves invoking operations on other Type Managers, then the controller uses the transaction facility to ensure the atomicity of the update. If the Type Manager is structured so that operations may be executed concurrently, the controller ensures that objects are not being modified by two operations simultaneously or read by one operation while being modified by another. Each type, in general, has its own set of constraints on the allowed order of execution of its operations on a given object. These constraints are supplied when the Type Manager is created.

2.5 Distributed Types

The reason for introducing the concept of distributed types in the system is to make transparent the distributed nature of an object that is logically viewed as a single object. The components of an object may be distributed by replication or partitioning. The transparency of the replicated or partitioned nature of an object is a convenient abstraction which makes updating and accessing of distributed and centralized objects identical.

A distributed type is an abstract data type whose concrete representation is distributed. For example, an abstract type called reliable-file might be implemented using physically distributed replicated copies of a file, or a global database might be implemented as a set of partitioned distributed components. The consistency and coordination among the distributed components of the concrete represention is specified in the type definition and enforced by the distributed Type Manager. Unlike the centralized objects, an occurrence of a distributed type does not have a unique host location, i.e., an object of a distributed type may "reside" at more than one host for reliability and performance reasons. An occurence of a distributed type is given a UID, the Type Manager then maps the operations directed to this UID into a set of operations, which are executed as a transaction, on the components that comprise the distributed object's concrete representation. This mapping can be done at any of the hosts where the distributed object is conceptually "residing". The operations defined for a distributed type are implemented as transactions.

3.0 STRUCTURE OF THE ZEUS SYSTEM

Zeus is essentially a collection of Type Managers (TMs); typically, many different Type Managers coexist on a host node. The core of the operating system consists of a set of Type Managers that support capabilities for defining new types and object instances in the system, authentication of users, naming environment for each user, and reliable process and transaction management functions. These system-defined Type Managers reside at every node in the system.

The lowest level of operating system at each node is called the kernel; the kernel virtualizes the resources at the host so that each Type Manager can be viewed as having its own virtual processor. The kernel supports interprocess communication, primary storage management, processor scheduling, interfaces to secondary storage devices, and UID generation. As shown in Figure 2, all Type Managers at a node execute over the abstract machine interface provided by the kernel. The kernel multiplexes the processor between the Type Managers; it also handles all interrupts due to storage devices and the communication devices.

3.1 Structure of the Zeus Kernel

The kernel consists of a task dispatcher and a number of interrupt handlers. The task dispatcher schedules the different Type Managers at its host node and handles their requests for resources. It also handles the restart of the system and initiation of the Type Managers. The resources managed by the kernel include volatile and non-volatile storage, the processor and the communication handler. The kernel interface consists primarily of three parts: invocation requests to other Type Managers, requests for unique numbers, and requests for resources. Storage management in the kernel is minimal. Storage is available in fixed sized blocks and the Type Managers request one or more of these blocks at any time. A Type Manager is solely responsible for the data it writes to the blocks of storage. The kernel keeps track of the ownership of blocks of storage. The routing of invocation requests to Type Managers is the major function of the kernel. Each call is an operation invoked against an object that is held by some Type Manager. Operation Switch, which is a component of the kernel, supports this function.

3.1.1 The Operation Switch

The function of the Operation Switch is to forward an invocation request to the appropriate Type Manager at the local or a remote node. These calls may be from a Type Manager or from the network driver. Each call contains the following information:

1. The extended UID of the object against which the call is invoked.
2. The extended UID of the process invoking the operation.
3. The extended UID of the principal on whose behalf the operation is being invoked.
4. The operation and a set of parameters.

The Operation Switch uses the host hint field of the target object's extended UID to determine whether the object is on the host or not. If it is, it uses the type unique number of the object to direct the call to the proper Type Manager. If the object is on another host, the Operation Switch instructs the Network Handler to send the call to the other host.

3.1.2 Unique Identifier Generation

The "type" and "instance" fields of an extended UID are unique numbers. Each of these unique numbers consists of three fields, the host identifier of the host at which they were generated, the incarnation number and the sequence number within an incarnation number. The kernel contains a component, called the SmallStepper, that generates some range of unique numbers. This component obtains the capability to generate multiple ranges of unique numbers from a distributed object called LargeStepper. The sequence number is obtained from the SmallStepper, which resides only in the volatile storage. The SmallStepper issues sequence numbers for a given incarnation number.

In a system where no failures can occur, each host will generate a monotonically increasing sequence of unique identifiers. If we permit failures, but stipulate that every host in the system has stable storage, then each host will store the next incarnation number, and as soon as it restarts on crash recovery, it will retrieve this number and write to stable storage the next incarnation number. Thus even though some part of a range of sequence numbers may not be generated, the hosts will generate a monotonically increasing sequence of unique numbers.

If we remove the assumption of stable storage on all hosts in the system, then hosts in the system can be divided into two classes: those that possess stable storage and those that do not. Each host with stable storage in addition to the SmallStepper has a process called the LargeStepper which together with the other LargeSteppers in the system generates new incarnation numbers. The algorithm used to do this is specified in a separate paper [DUTT83].

3.1.3 Network Handler

This component provides a simple datagram level of transport mechanism between different kernels. It interfaces with the Operation Switch. The invocation requests for remote nodes are handed over by the Operation Switch to the Network Handler, which has the responsibility for delivering it to the Operation Switch at the destination host. Similarly the response messages are returned from the server to the invoker by the network handler via the Operation Switch.

3.1.4 Kernel Initiator

The kernel initiator has two functions. The first function is to restart a host when it recovers from a failure. The second is to initiate a task. Both tasks require a certain amount of housekeeping. Host recovery implies the setting up of tables for the dispatcher of the kernel, using the log for the Type-Type Manager to create, delete, or modify the Type Managers on the host, and obtaining a new incarnation number and the SmallStepper sequence number. After the above actions are successfully completed, the initiator can hand control to the task dispatcher.

3.2 System-Defined Type Managers

As mentioned previously, Zeus is a set of Type Managers whose members may potentially change dynamically as Type Managers are created, deleted, and modified. There is, however, a subset of Type Managers called the System Type Manager which perform the essential services provided by the kernel of a conventional operating system. In this section, the Type Managers for these system types are defined. The following are the System Type Managers which exist at each node in the system.

(1) Type-Type Manager
(2) Process/Transaction Manager
(3) Principal and Authentication Manager
(4) Symbolic Name Manager
(5) Program Manager
(6) Message Manager

The functions provided by these Type Managers along with their structures are described below. Each of these Type Managers is considered as an object of distributed type; an instance of each of these Type Managers resides at every node. The distributed Type Managers for a given type function cooperatively to provide the abstraction of a single system-wide Type Manager.

3.2.1 Type-Type Manager

The definitions of new Type Managers is introduced in the system by using the mechanisms supported by a system-wide object called the Type-Type Manager. Thus, the Type-Type Manager implements functions to create, alter, delete and replicate Type Managers. The definition of the Type-Type object given here is an adaptation and extension of the Type-Type concepts originating in the HYDRA [WULF81] operating system. The facilities provided by the Type-Type Manager include an explicit command on where to locate copies of a Type Manager.

Type managers are active objects. At any point in time, one or more copies of the Type Manager for a given type may be active. By active we mean that either within the Type Manager calls against its object instances are in progress, or that some of the fucntions it implements have invoked calls on some other Type Manager and are waiting for a return. This complicates the Type-Type manager because it must ensure that all copies of a Type Manager are in a quiescent state and will stay in that state before an operation can be invoked against that Type Manager. However, we believe that operations to modify existing Type Managers will be quite infrequent, therefore, schemes based on global synchronization can be used for consistency management.

3.2.2 Process/Transaction Manager

Processes and transactions are active objects in the system through which a user carries out operations in the system. Transactions are atomic processes, i.e. they have an "all or nothing" property. The transaction facility with its atomic property provides a powerful mechanism for reliable operations. A transaction either commits or aborts on termination, and if it aborts then no trace of its execution is left. On the commitment of a transaction, all updates made by it are permanent.

We require that a process must invoke a transaction in order to modify permanent shared objects in the system. The changes to an object are recorded as new versions of the object. New versions of the object are committed to becoming permanent at the end of a successful completion of the commit protocol among the invoked transaction, the invoking process, and the Type Managers of the modified objects. Uncommitted versions are discarded on explicit abort commands issued by the transaction process or on timeout due to inactivity.

Processes and transactions can establish recovery points by checkpointing. Such points are used for the purpose of rollback and restart of a process or transaction. Checkpointing is the selective saving of versions of process or transaction objects. Note that with the above scheme for checkpointing, only the state of the process (or transaction) object is saved; the states of objects modified by that process are not saved in the checkpoint. This approach may create problems for error recovery since not all state changes of the process are recorded with the checkpoint. However, one must remember that all updates made within a transaction to permanent objects via their Type Managers are saved on the stable storage as uncommitted versions. It is, therefore, necessary to exersize some discipline in using checkpoints and atomic transactions. The following discusses how checkpointing can be used correctly to support recovery.

The first problem that we want to address is how one guarantees correct rollback recovery. One requirement for correctly implementing rollback is that the object manager for transactions must maintain for each transaction a list of UIDs of the objects that are affected by the transaction. The reason for this is that in the event of a rollback, it requires that all changes to objects made by the transaction after the checkpoint be discarded. The list of UIDs of the objects that are affected by the transaction provides a means for notifying these objects to discard the aborted versions. This list is also used at the end of the transaction to conduct the commit protocol. The discussion in the preceding paragraph implies that for implementing rollback, timestamps must be recorded with each checkpoint and each version of objects. This requirement stems from the fact that there is not a one-to-one mapping between process or transaction checkpoints and versions of objects affected by them.

The second problem is the interaction between process checkpointing and commitment of a transaction invoked by the process after that checkpoint. Suppose a process crashes after committing a transaction. In such a case the process restarts from its last checkpoint, but the transactions that have been committed since the establishment of this checkpoint are not undone. Thus some committed transactions might be executed more than once due to the restart. If a transaction is nonidempotent, i.e. multiple executions of the transaction produce different results, a problem may arise in error recovery since rollback of the process may cause a committed transaction to be executed again. One solution to this problem is to always force the invoking process to checkpoint concurrently with the committing transaction. Checkpointing is part of the commit protocol; if the protocol determines to abort, the checkpoint is discarded. With this mandatory checkpoint, rollback recovery of a process can avoid undesirable repetition of transaction execution. However, this may cause too frequent checkpointing of the invoking process. Therefore the second solution is to make a checkpoint of the invoking process an option that is to be specified at the time of invoking a transaction. Apparently this checkpoint is not required for idempotent transactions to guarantee correct execution. However, a process invoking an idempotent transaction may elect to force a checkpoint during the transaction commit protocol for efficiency reasons. For example, if a transaction requires extensive computation compared to checkpointing the invoking process, and if the possibility of a failure is significant, it may be desirable to have a checkpoint as described above. The decision of when to checkpoint is left to the process that invokes the transaction.

The Process/Transaction Manager also supports nesting of transactions; such nested transactions can execute concurrently with the parent transactions. The nested transaction facility provides the users mechanisms to introduce concurrency within a transaction. The commitment of a nested transaction is dependent on the commitment of the parent transaction.

3.2.3 Principal and Authentication Manager

The object protection system in Zeus depends on the ability of the individual Type Managers to identify any process which requests an operation be performed. In addition, the Type Managers need to be able to determine the ultimate initiator of the action which resulted in such an invocation request. We call these initiators of actions principals. Principals are permanent objects in Zeus and they are the only objects which carry the authority to perform computations involving other objects. When a new process is created, it is "owned" by a single principal and it retains this principal association throughout its lifetime.

The two fundamental problems of the protection system, authentication and authorization, both involve principal objects and the association of processes to principals. The problem of authorization, that is determining on whose behalf a given process is currently working, is a fairly simple matter since each process is always working for a single principal only. When a process invokes an operation on a Type Manager, the information regarding its UID and principal association is transported onto the virtual machine of the target Type Manager. In this way, the principal which owns a particular process is always known by any Type Manager on which it makes invocation requests. In addition, since process identifiers are transported to and from Type Manager machines by system code, a process is unable to forge its own principal association to gain access to objects its real principal is not authorized to access.

During login, a user is first asked to identify himself by giving his unique principal symbolic name. The login process (also called the Authentication Manager) tries to find a principal object containing the same symbolic name. The principal object contains all the pertinent information about that user. The user's password is stored with the principal object, allowing the Authentication Manager to perform necessary authentication checks. Two other pieces of information regarding the user are maintained within the principal data object. One is the unique identifier (UID) of the user's symbolic name context, which is described in the next section. The other is the UID of the command interpreter or shell program of the logged-in principal.

Since the authentication manager must find a principal object given only its symbolic name, it follows that this name must be unique. In order to make it convenient for unique names to be assigned to principals, Zeus has the concept of a working group (WG). Working groups are used to form a strict hierarchy of principal names. This hierarchy of names is similar to that used in the Multics system. They contain members which may be either principals or other working groups. The root working group has a null name and is called the null working group. The unique name of a principal or working group is formed by concatenating the name of the principal or WG with the names of all of its containing working groups. This hierarchical structure also forms the basis for other symbolic names in the system.

3.2.4 Symbolic Name Manager

To provide user convenience, an object can be given a symbolic name that is used when referencing that object. A user in the system should be able to use symbolic names within its context independent of other users. For example, the same symbolic name can be used by different users to refer to different objects. Similarly, different symbolic names can be used by different users to refer to the same object. The Symbolic Name Manager maintains the mapping between a symbolic name for an object and that object's UID. The mapping function is many-to-one in that several symbolic names may be mapped to one object UID. The symbolic names within a context must be unique.

The objects which are managed by the Symbolic Name Manager are symbolic name contexts, where a context object contains the above mentioned mapping. A context may be viewed as a private directory of relative symbolic names. Each principal is given a context when the principal is created. It is initialized with the symbolic name to object UID mappings of certain system objects which a principal must know in order to function properly. The Symbolic Name Manager maintains a data base that contains the context objects and the current state of the context operations. In the event of a failure the data base provides the recovery of the state of the Symbolic Name Manager and the recovery of the context objects.

Contexts, like other objects, may be shared among principals having the proper access and are part of the mechanism by which principals may share objects. If principal A wishes to share object X with principal B, A must give B access rights to X and must also give B the UID of X. When A shares its context with B, B is able to obtain the UID of X through an agreed upon symbolic name for X. It is important to note that sharing a context in no way enhances or alters the access rights to any of the objects whose UIDs are in the shared context. Access to an object is still coordinated by its associated Type Manager.

3.2.5 The Program Type Manager

The Program Type Manager is the repository of both program text and object code. Program text is defined to be a text object that compiles correctly. Thus, the creation of a program object requires the user to supply the Program Manager with a correct program or a separately compilable unit of a program. The Program Type Manager in addition to its function as a repository acts as a builder of programs. Thus, a user can call upon the program Type Manager to build a new program from some specified components. This linking function of the Program Type Manager is useful to the system to build new user types. A program object is defined to be a collection of versions of a single program. The criteria for retaining program versions in the system are defined by the users.

3.2.6 Message Type Manager

The Message Type Manger provides for the synchronous and asynchronous exchange of message between processes. At the time a message is created, the sender can specify the reliability class for that message. The reliability class of a message reflects its availability to the receiver in the face of one or more host failures in the network. At the low end of reliability there are volatile message objects that disappear upon host failure (if the object resides on the failed host). At the high end of reliability stable message objects have a replication factor of n where n is the number of hosts in the network. Two additionalintermediate reliability classes exist.

Interprocess communication may occur between processes that are local to a host, or remote. In either case message operations are performed by the message type manager local to the host of the invoking process. Any remote communication required by the operation is done by peer Message Type Managers and is unseen by the processes involved.

4.0 CONCLUSIONS

In this paper we have presented an overview of the Zeus distributed operating system which is suitable for highly reliable applications. Zeus is an object-oriented system which is novel in the sense that it integrates many of the conventional database management functions into the operating system. Recovery and synchronization are transparent to the application programmers. The software for this system looks no different than the conventional software because of the remote procedure call mechanism which makes accessing of remote and local objects identical. This distributed operating system is basically a collection of system and application defined object managers (also referred to as Type Managers because they manage the objects of a specific type). In this paper we have described some system defined Type Managers that provide the essential facilities to the application developers for creating their own Type Managers. This paper has presented certain principles that have not been tested yet in a real implementation; there still remains a significant amount of research to be done to demonstrate these ideas in a real system.

REFERENCES

[AKER83] Akers, L., W. Bevier, R.M. Cohen, D.I. Good, L.M. Smith, M.K. Smith, "Formal Proof of Recovery Mechanisms," Institute of Computing Science, University of Texas at Austin, Austin, Tx 78712, October 1983.

[BIRR82] Birrell, Andrew D., Roy Levin, Roger Needham, Michael Schroeder, "Grapevine: An Exercise in Distributed Computing," Communications of the ACM, April 1982, pp. 260-273.

[BROW83] Browne, James C., "Structuring Strategies for the Design and Implementation of High Integrity Systems: A Vertically-Structured Approach," Research Report, Information Research Associates, Austin, Texas 78705, 1983.

[COHE75] Cohen, Ellis, David Jefferson, "Protection in the Hydra Operating System," Fifth Symposium on Operating Systems Principles, 1975, pp. 141-160.

[DUTT83] Dutton, Jim, Fernandes, Vincent, "A Scheme for Reliable Generation of Unique Sequence Numbers in a Local Area Network," Technical Report, Information Research Associates, Austin Tx, September 1983.

[LISK82] Liskov, B., Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Ninth Annual Symposium on Principles of Programming Languages, January 1982, pp. 7-19.

[MOSS81] Moss, J., Elliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, Massachusetts 02139, April 1981.

[NELS81] Nelson, B. J., "Remote Procedure Call," Technical Report-Department of Computer Science, Carnegie-Mellon University, May 1981.

[OPPE81] Oppen, Derek C., Yogen K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, Palo Alto, California 94304, October 1981.

[WALK83] Walker, B., et al., "The LOCUS Distributed Operating System," Ninth ACM Symposium on Operating System Principles, October 1983, pp. 49-70.

[REDE80] Redell, David D., et al., "Pilot: An Operating System for a Personal Computer, "Communications of the ACM, February 1980, pp. 81-91.

[SCHA83] Schantz, R., et al., "Cronus: A Distributed Operating System - Interim Technical Report No. 2, "RADC Report No. 5261, February 1983.

[SHRI81] Shrivastava, S. K., "Structuring Distributed Systems for Recoverability and Crash Resistance," IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981, pp. 436-447.

[SHRI82] Shrivastava, S.K., F. Panzieri, "The Design of a Reliable Procedure Call Mechanism," IEEE Transactions on Computers, July 1982,

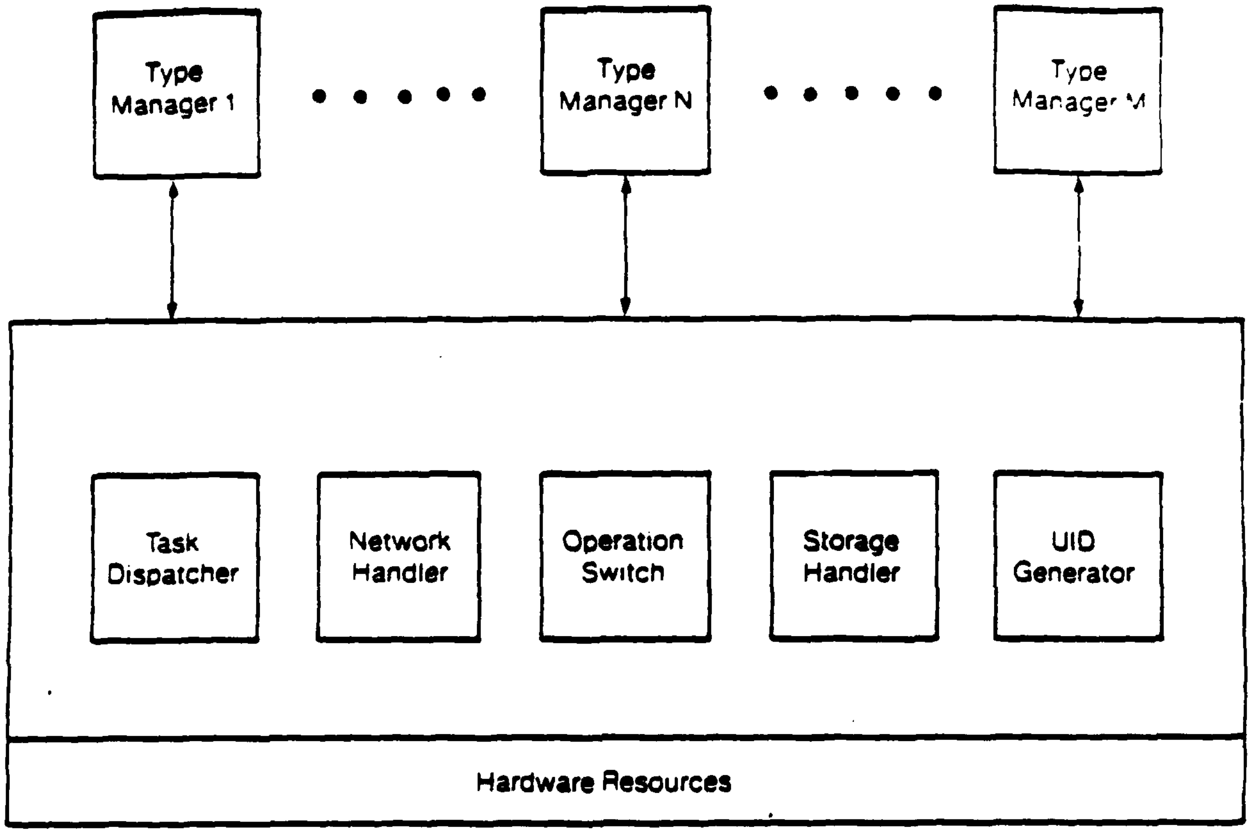
[STON81] Stonebraker, Michael, "Operating System Support for Database Systems," Communications of the ACM, July 1981, pp.412-418.

[SVOB81] Svobodova, L., "A Reliable Object-Oriented Data Repository for a Distributed Computer," Eighth Operating Systems Principles Conference, 1981, pp. 47-58.

[TRIP83a] Tripathi, A. R., P. S. Wang, "An Object-Oriented Design Model for Reliable Distributed Systems," Third Symposium on Reliability in Distributed Software and Database Systems, October 1983.

[TRIP83b] Tripathi, A. R., W.T. Wood, "A Formal Model to Prove Global Properties in Distributed Systems," Technical Report, Corporate Computer Sciences Center, Honeywell Inc., Bloomington, MN 55420, September 1983.

[WULF82] Wulf, William A., Roy Levin, Samuel P. Harbison, HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, 1981.



Structure of a Zeus Host

FIGURE 2

SOME PERFORMANCE MODELS OF DISTRIBUTED SYSTEMS

Vincent Fernandes
J. C. Browne
Doug Neuse
Rajkumar Velpuri
Information Research Associates
Austin, Texas

ABSTRACT

Performance models of a distributed, real-time command and control system are presented, including models analyzing fault-propagation and associated fault recovery strategies. The techniques and tools described here are applicable to the design and analysis of distributed systems in general and are ready and available for use today by modeling practitioners.

1. INTRODUCTION

Truly distributed systems are now finally beginning to appear in substantial number in information management applications. Distributed systems introduce a new set of implementation techniques and mechanisms and thus a new set of performance limiting factors. These factors include 1) the additional interfaces and mechanisms that integrate the local processing at a given site into global resource environments as well as 2) the actual costs of communication and data movement.

This paper describes an integrated practical framework for introducing these interfaces and mechanisms into the performance modeling of truly distributed processing systems. The paper begins with a conceptual description of techniques for distributed computing and the added processing that results. This additional processing can be viewed in a hierarchical fashion: The lowest level is a logical representation of a network. The representation which we use here, for the sake of concreteness, is clusters of hosts connected by long distance links. Each cluster is structured internally using an Ethernet [MET76]. The next higher level is the interface of Remote Procedure Calls (RPCs) and messages to the logical network defined by packet transmission. RPCs couple normal local processing in higher level languages with remote resources. Message based processing may also be made visible at the application level, but RPCs are the mechanisms that most naturally introduce remote processing into normal procedural programs for information management applications.

These distributed processing system elements are described in a format appropriate for developing performance models. This description is then developed in terms of information processing graphs (IPGs), a diagrammatic framework for combining workload, system structure, and hardware configurations. The IPGs can be thought of as generic models from which specific evaluatable models can be constructed in terms of some simulation (or analytic) modeling language. The modeling language used in this project was the Performance Analyst's Workbench System (PAWS) [IRA84]. A limited number of copies of the PAWS models described here are available upon request. The models described here were developed in a top-down, hierarchical manner in conjunction with the top-down design of the distributed system. The IPGs shown here reflect this hierarchical development: the initial IPGs document, the flow of information at the highest levels in the system, and as our explanation proceeds, each component in the high-level IPGs is expanded until an appropriate level of detail is reached. It is anticipated that these modeling techniques and tools can be used to extend performance modeling into the realm of distributed systems in a practical way.

This representation of the performance of distributed systems has been developed in the context of a project for determining the costs of introducing very high reliability into distributed systems. This project is named ZEUS [BRO84] and is based on the concept of object-oriented programming. Each entity in the system belongs to some object type, and an object instance encapsulates the data structure of the object and the operations that can be invoked against it. Objects are identified by a system-wide unique identifier: operations can be invoked against objects locally or from a remote host. In this context the need for a remote procedure call is obvious. The interest in this project is focused upon the relative costs of system execution with and without reliability mechanisms. The system we are modeling has not, in fact, been implemented. The techniques we define for modeling distributed systems do, however, appear to offer a broadly applicable framework upon which to extend performance models of a traditional structure directly to distributed resource environments.

Section 2 briefly describes the IPG notation and the PAWS language. Section 3 describes the physical execution environment involved in this project. Section 4 illustrates how fault-free distributed systems can be modeled, including models for the RPC, hosts, and communication. Section 5 introduces faults into physical resources and demonstrates how these can be propagated upwards to the users of these resources, thus permitting recovery strategies to be modeled. The RPC can be viewed as an example of distributed system usage. Other applications could use the physical resource models in a similar fashion. Thus, we present here a set of techniques that can be used to model a wide variety of distributed system usage.

2. INFORMATION PROCESSING GRAPHS AND PAWS

The performance models described here were developed using Information Processing Graphs (IPGs) and the Performance Analyst's Workbench System (PAWS). This section briefly describes IPGs and PAWS; for a complete description see [IRA84].

The use of IPGs and PAWS is diagrammed in Figure 2.1.

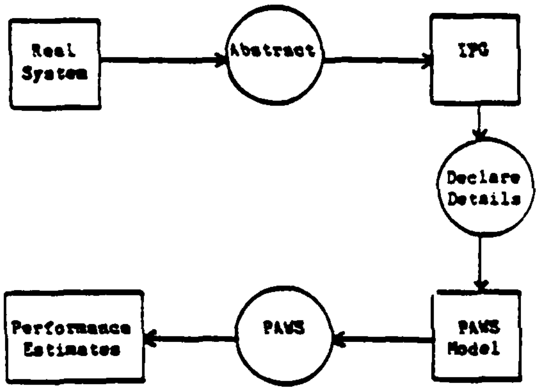


Figure 2.1. The Use of IPGs and PAWS

The modeler begins with an existing or to-be-designed real system and abstracts from that system a high-level picture of the flow of information in the system. This picture is called an IPG. The details of each part of the picture are then declared in the PAWS simulation language to obtain a PAWS model, which is evaluated by the PAWS simulator to obtain performance statistic estimates such as response times, throughputs, utilizations, etc.

2.1 INFORMATION PROCESSING GRAPHS

An IPG diagrams the flow of information from resource to resource in an information processing system (e.g., see Figure 2.4). Information processing systems, such as computing, communication and office systems, can be thought of in terms of work stations or nodes at which information is processed. In a computing system the nodes may represent central processors, disk units, device controllers, etc. Edges have labels denoting the form (transaction category, transaction phase, branching probability) of information flow along the edge. Information flows in discrete units called transactions. A transaction in general represents the data on which the nodes of the information processing graph operate and in particular may represent a job, program, task, scheduler, message process, person, or any such entity useful to the modeler. A transaction gets processed in some manner at each node and, upon completion of processing, moves along the direction of an edge to some other node for additional processing. Several transactions may be simultaneously active (being operated upon) at the various nodes of the IPG.

We associate a category and a phase with each transaction. The category of a transaction is a name (denoted by any string of alphanumeric symbols) and is permanent: a transaction has one unique category throughout its lifetime. The phase of a transaction (denoted by an integer) may be changed as the transaction progresses through the IPG. The processing of a transaction at a node and the routing behavior of a transaction from node to node in general depend on that transaction's category and phase. The general form for denoting a transaction's category and phase is <category name, phase number>.

There are five classes of nodes used in the IPG notation: 1) resource management nodes, 2) routing nodes, 3) arithmetic nodes, 4) INTERRUPT nodes, and 5) USER nodes. Each class of nodes is discussed briefly below.

(1) Resource management nodes represent system resources (processors, memory, communication links, disks, copying machines, people, etc.). A transaction normally requests the use of certain resources and may have to queue (wait) for a resource if the request cannot be fulfilled immediately. Thus, resource nodes have queues associated with them. Resources may be classified as active resources or passive resources.

Conceptually, an active resource is something that acts or works, such as a processor or disk unit. An active resource is represented by a SERVICE node in PAWS. A transaction arriving at a SERVICE node requests the use of the resource for a specified amount of time (usually drawn from a specified service time distribution). If the resource is being used, the transaction must queue (wait) until it is scheduled for service according to the queueing discipline specified for that SERVICE node. After receiving service the transaction exits the node along some edge to another node. Figure 2.2 shows a portion of an IPG representing a SERVICE node named CPU. The circle represents the processor and the open box or square represents the queue for waiting transactions.

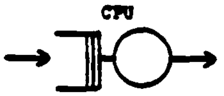


Figure 2.2. SERVICE Node CPU

A passive resource doesn't itself do any work but is something that must be possessed by a transaction to do work. Memories, buffers, and control points are examples of passive resources. Passive resources usually occur in groups; for example, memory may be regarded as a group of pages. The amount of time a passive resource is held by a transaction is not specified by a service time distribution. After acquiring a passive resource (such as memory), a transaction typically uses one or more active resources (processors, disks, etc.) before releasing the passive resource. Thus, a passive resource is represented in an IPG by two nodes: one at which the resource is acquired and one at which the resource is released..

There are two types of passive resources: TOKENS and MEMORIES. TOKENS are acquired at ALLOCATE nodes and released at RELEASE nodes. MEMORIES are acquired at GETMEM nodes and released at RELMEM nodes. TOKENS may be used to model input and output buffers, channels, pages, domains or control points, communication links, and other passive resources. Typically, a separate token is used to represent each resource (buffer, page, etc.), and these tokens are partitioned into type with one token type for each type of resource (input buffers, main memory pages, etc.). MEMORIES are used to model contiguously addressed passive resources such as main memories, extended core storage, and disk space. Associated with each memory is a memory management scheme according to which blocks of memory are allocated to transactions.

TOKENS do not have to be RELEASEd to the node at which they were ALLOCATEd. At a RELEASE node a transaction may specify any ALLOCATE node to which the tokens are to be released. TOKENS may be created at CREATE nodes and destroyed at DESTROY nodes. Figure 2.3 shows a portion of an IPG in which transactions a) acquire BUFFER tokens at the ALLOCATE node named GET, b) create BUFFER tokens for GET at the CREATE node named MAKE, c) destroy BUFFER tokens for GET at the DESTROY node named KILL, and d) at the RELEASE node named PUT, release BUFFER tokens for the ALLOCATE node named GETMORE.

Associated with each node at which resources (active or passive) are acquired is a queueing discipline, i.e., the discipline according to which transactions enqueue if the resource is not available immediately.

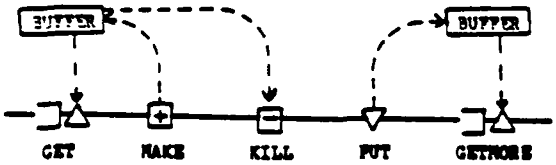


Figure 2.3. ALLOCATE, CREATE, DESTROY, RELEASE Example

The rate at which an active resource processes information or the ability of a transaction to acquire a passive resource may depend on events occurring in other parts of the system. In such cases, a transaction at a SET node may request a modification to the service rate or power of a SERVICE, ALLOCATE, or GETMEM node.

(2) Routing nodes may be used to create and destroy transactions and to alter transaction flow through the system. There are six types of routing nodes: SOURCE, SINK, FORK, JOIN, SPLIT, and BRANCH nodes. At a SOURCE node, transactions are created (arrive) periodically according to a user-specified interarrival time distribution. At a SINK node transactions disappear from the system forever. A transaction may spawn a number of children transactions at a FORK node, and the children may coalesce at a JOIN node to recreate the parent. FORK and JOIN nodes are useful for modeling the synchronization of concurrent processes. A transaction may create a number of SIBLING transactions at a SPLIT node - much like a JOIN node. The transactions created at a SPLIT node may, for instance, be used to model the operation of message communication. BRANCH nodes may be used to facilitate the specification of branching (routing) probabilities and to collect statistics.

Figure 2.4 illustrates the use of the routing nodes. Each transaction enters the system at the source node START and proceeds to the fork node TFORK, where the transaction creates two children and waits for the children to join.

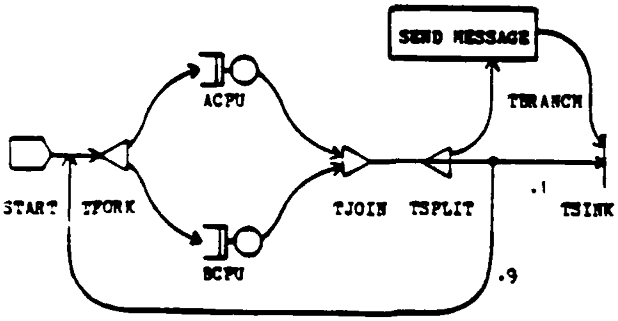


Figure 2.4. Routing Node Example

Each child requests and uses a processor (ACPU or BCPU) before proceeding to the join node TJOIN. As soon as both children of a transaction reach TJOIN the children disappear and the parent (still waiting at TFORK) replaces its children at TJOIN and proceeds from TJOIN to the split node TSPLIT, where a sibling transaction is created. The newly-created sibling proceeds to send a message before leaving the system at the sink node TSINK. The original sibling (the transaction that started at START) travels from TSPLIT to the branch node TBRANCH, from which it goes back to TFORK with probability 0.9 or to TSINK with probability 0.1.

(3) Arithmetic nodes are used to carry out computational steps and to modify simulation variables. There are two types of arithmetic nodes: COMPUTE nodes and CHANGE nodes. Each transaction has local variables associated with it. In addition, the network has some global variables associated with it. A COMPUTE node is used for assignment of values to and conditional operations on these variables. A CHANGE node is used to change the phase of a transaction (probabilistically). In Figure 2.5, a transaction arriving at the compute node ACONP increments a global variable named COUNT and proceeds to the CHANGE node named SWITCH, which changes the transaction's phase from 1 to 2 or 2 to 1.

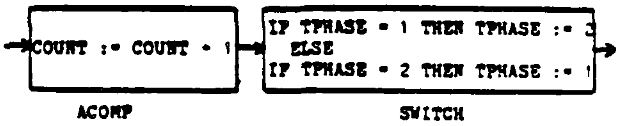

COUNT := COUNT + 1
IF TPHASE = 1 THEN TPHASE := 2
ELSE
IF TPHASE = 2 THEN TPHASE := 1
ACOMP
SWITCH

Figure 2.5. Arithmetic Nodes

(4) An INTERRUPT node is used by one transaction to interrupt the processing of another transaction. The interrupted transaction immediately departs from the node where it was interrupted with a new phase assigned to it by its interrupter.

(5) A transaction arriving at a USER node invokes a user-written FORTRAN subroutine that has access to the global variables and the transaction's local variables. The USER node facility makes PAWS an extensible system.

2.2 THE PAWS LANGUAGE

The PAWS language is declarative rather than procedural: the user simply declares the characteristics of the system being modeled as opposed to coding detail simulation algorithms. An example CPU node definition in the PAWS language is shown in Figure 2.6.

```
CPU
 TYPE                 SERVICE
 QUANTITY             1
 QD                   RRFQ 10.0
 REQUEST              <BATCH,ALL> HYPER(10.0,14.1);
```

Figure 2.6. Example Node Definition in PAWS

Here, the name of the node is CPU, the node type is SERVICE, there is 1 server, the queueing discipline is round-robin fixed quantum with a fixed quantum of 10 time units, and all BATCH transactions regardless of phase request service times drawn from the hyper-exponential distribution with mean 10.0 aud standard deviation 14.1.

3. PHYSICAL EXECUTION ENVIRONMENT

Any network software must execute on an underlying system. The physical system involved in the ZEUS project consists of clusters of hosts that are connected by long distance links. The hosts in a cluster are connected using a CSAN/CD local area network such as Ethernet. We assume that a message is broken up into packets at its source and re-assembled at its destination. This implies a universal packet structure that all the hosts in the network understand. The network sub-model described in section 4.1 assumes this underlying physical execution environment. Nevertheless, the general principles presented in this paper are entirely independent of that environment.

4. THE REMOTE PROCEDURE CALL

In this section we develop a model for a Remote Procedure Call (RPC) from the application level to the level of packets transmitted through the network. The RPC model of distributed programming permits its users to access remote programs using procedure or function call semantics. Thus, an RPC can be viewed as a call-response message pair. A number of schemes, including the one presented here, choose to incorporate acknowledgements for calls and responses. Figure 4.1 shows the IPG for an instance of a call. There are three nodes: SETCALL, RPC, and SEEVAL. The node RPC is a complex (i.e., high-level) PAWS node and will be expanded later in this section. A complex, high-level node is denoted in IPGs by double vertical end bars. The nodes SETCALL and SEEVAL are PAWS COMPUTE nodes that set the call parameters and inspect the results of the call. The call parameters are call source, call destination, service required and call size. These parameters will determine the services required by the call enroute and at its source and destination.

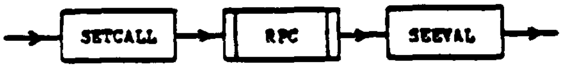


Figure 4.1. IPG for an RPC

Figure 4.2 shows the sequence of messages exchanged during an RPC. The caller can proceed after a response has been received. Likewise the callee can continue after the response is sent.

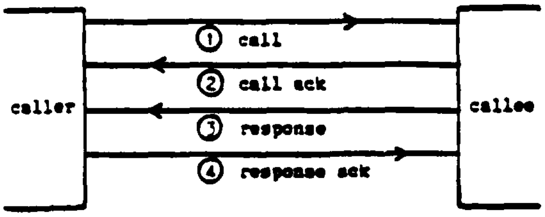


Figure 4.2. The Message Flow for RPC

In Figure 4.3 we develop the IPG for the message sequence of Figure 4.2. This IPG is an expansion of the complex RPC node of Figure 4.1. All of the transactions that flow in this IPG are of the category CALL, thus permitting a simple structuring of the system. The CALL category uses the complex node NET to deliver messages from host to host. The complex node TMD is used to process calls at a host. The node SETPARM sets the parameters of a message (i.e., allows the source, destination and type of message to be set). Although not shown in Figure 4.3 the different phases of the category CALL are used to distinguish between the different messages sent to achieve the RPC.

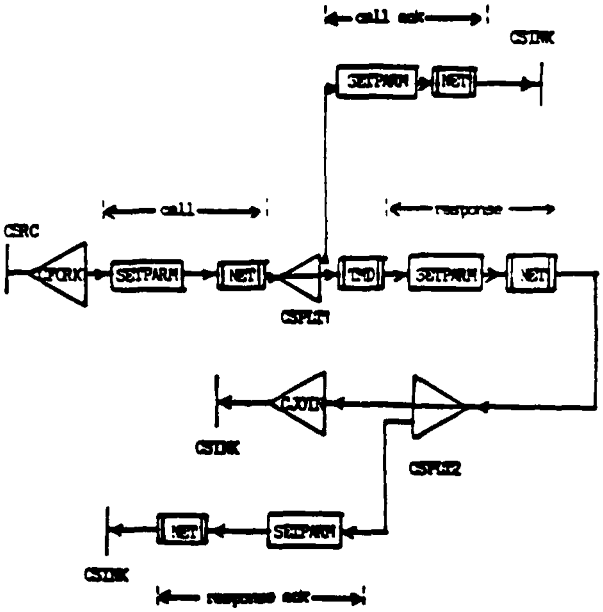


Figure 4.3. IPG for the RPC Messages

A transaction entering the RPC Sub-model creates a child transaction at the CFORK node. The child transaction represents the remote procedure call; the parent transaction remains in limbo at CFORK until the child arrives at CJOIN. The child proceeds to SETPARM, where the message parameters (source, destination, and type of message) are set, and then to NET, which delivers messages from host to host and is described in detail later. On exit from NET, the child has arrived at its destination and proceeds to CSPLT1, where it creates a sibling transaction to model the call acknowledgement. Immediately after this split (i.e., in parallel with its newly created sibling) the child proceeds to TMD, which models the call processing. On exit from TMD, the child sets its message parameters at SETPARM and invokes the NET sub-model to model the response, after which the child creates another sibling transaction to model the response acknowledgement. Immediately after this split, the child performs a join operation at CJOIN, which removes the parent from limbo at CFORM and sends the parent to CSINK to depart the RPC sub-model.

We next develop the complex node NET.

4.1 THE NETWORK SUB-MODEL

This section presents an expansion of the NET complex node in Figure 4.3 and assumes a physical execution environment as described in section 3. The overall IPG is given in Figure 4.4. Briefly after some computation at the source host the message is broken up into packets based on its size. The IPG uses the fork-join pair PF and PJ to model the packetization and re-assembly of the message. The computation done when the message is sent and received is modeled by the complex node COMP which represents host computation. Each generated packet goes through the network via the complex node PKTSND which is described next. Notice that within this IPG there are three categories: before MF and after MJ there is the category CALL, between MF and MJ there is a category MSG, and between PF and PJ there is a category PKT. This was done because we felt that the packetization and re-assembly processes are a part of the message level.

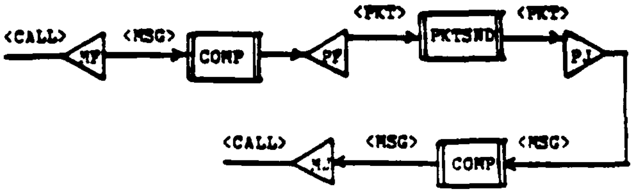


Figure 4.4. Sending a Message

The PKTSND complex node describes how packets are routed through the network. Each hop through the network can be through a long distance link (LDL) or a local area network. We define statically for each pair of clusters in the network a set of paths that will permit transmission from the source cluster to the destination cluster. Typically a packet will go from source to the first gateway via a LAN and then via some combination of LDLs and LANs to the destination host's cluster. Finally the packet will go via the destination host's LAN to the destination host. The PKTSND node is expanded in Figure 4.5. The node INILC sets up the path packet whereas NXTLG determines the next leg of the path the packet will take.

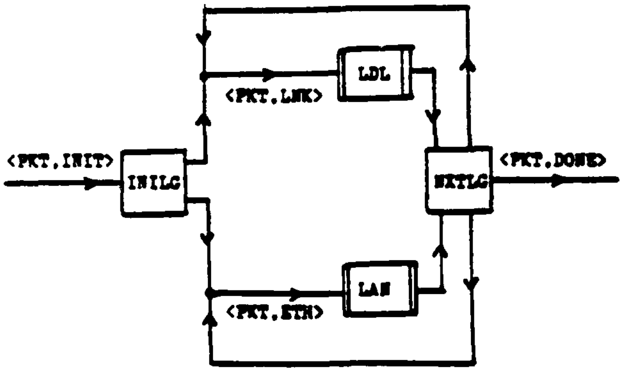


Figure 4.5. The PKTSND Complex Node

Each PAWS transaction carries with it information identifying the specific link or local area network the transaction will visit, along with the source and destination gateway hosts on the links or local area networks. We next develop the models for the long distance links and the local area networks.

4.2 A MODEL FOR A LONG DISTANCE LINK

The long distance link at its simplest can be modeled as a first come first served queue. We have chosen not to model packet acknowledgements at this level though this can be included in the model. The IPG is shown in Figure 4.6. There LNK is a FCFS queue whereas the two COMP complex nodes represent computation at the source and destination of the link.

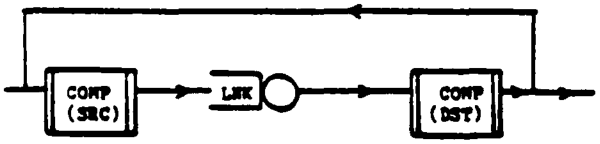


Figure 4.6. Model for the Long Distance Link

The node LNK could simply be a delay since we assume that packets are all the same size. Finally to model packet retransmission we have an edge in the IPG that a transaction takes should the packet's data be corrupted.

4.3 A MODEL FOR THE LOCAL AREA NETWORK

The packet that enters the LAN sub-model knows its source, its destination and the Ethernet on which it must be broadcast. Further, at any given time a host can attempt to broadcast only a single packet on the LAN. Internal to the LAN the interactions are more complex. These are therefore developed in the complex node ETHER.

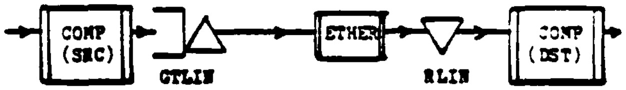


Figure 4.7. Model of the LAN

The COMP nodes again represent computation and the GTLIN and RLIN represent usage of a token to ensure that any host will attempt to broadcast only one packet at a time. We next develop a model of the ETHER complex node; Figure 4.8 shows the corresponding IPG. We note that QL (<service_node>) is the current value of the queue length at that service node. In this model there are three service nodes: B-del, P-del, and T-del; and two compute nodes: SETBACK and SETROUT.

The basic notion behind a CSMA/CD protocol is that a broadcast has two phases: propagation and transmission. During propagation, packet collisions can occur. During transmission, the carrier sense mechanism causes the other hosts to hold their packets for some back-off period.

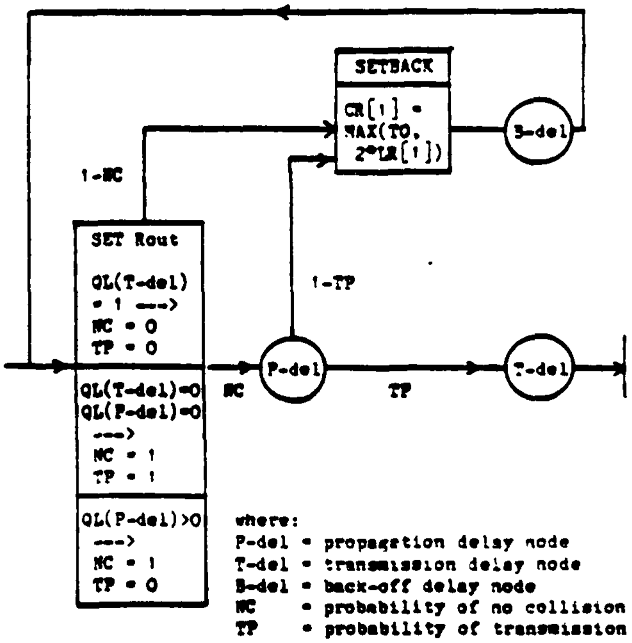

SETBACK
CR[1] = MAX(TO, 2**LR[1])
B-del
1-NC
SET Rout
QL(T-del) = 1 ---> NC = 0 TP = 0
QL(T-del)=0 QL(P-del)=0 ---> NC = 1 TP = 1
QL(P-del)>0 ---> NC = 1 TP = 0
1-TP
P-del
NC
TP
T-del
where:
P-del = propagation delay node
T-del = transmission delay node
B-del = back-off delay node
NC = probability of no collision
TP = probability of transmission

Figure 4.8. Details of CSMA/CD protocol

If a packet (transaction) arrives with QL(P-del) and QL(T-del) equal to zero then the net is not busy so the branching probabilities NC and TP are set to one. This is the normal case. The packet will wait until its propagation delay P-del is over. Then if no collision has occurred, the packet will proceed to T-del (because TP is still set to 1) and will then depart the CSMA/CD protocol sub-model. If a collision has occurred while the packet is at P-del then TP will have been reset to zero, so the packet will go to SETBACK to compute its back-off delay and then to B-del wait for that time, after which it will retry transmission.

If a packet arrives when QL(P-del)>0 then one or more packets before it are undergoing a propagation delay. These previous packets must be blocked from going on to transmission delay. Therefore, the newly arriving packet sets TP to zero and proceeds to P-del so that the residual effect of its broadcast will be felt by any subsequent packets. After waiting at P-del the colliding packet will leave (since TP is zero) to calculate its back-off delay and then to the back-off delay node B-del after which it retries transmission.

If a packet arrives when QL(T-del)>0 (it can only be 1) then the packet sets NC to zero and thus routes itself to the backoff delay calculation node. From there it goes to the backoff delay node and then to retry transmission.

4.4 THE MODEL OF A HOST

The handling of calls, messages, and packets needs computation. The corresponding transactions in our model evoke computation at the hosts they visit. The computations have been represented as complex nodes in the previous IPGs. This subsection presents an expanded model of such computation.

We model a host as being a set of physical resources and a set of processes each of which executes one of a fixed set of computations. A PAWS transaction that wants to evoke a computation on a host will instantiate a process on that host. This process is modeled by a PAWS transaction of category COMPT that will execute one of a set of computations based on the parameters set by the instantiator. The processes that execute on the same host are modeled by SIBLING transactions, the advantage of which will be pointed out in the next section.

We assume the physical resources of a host to be a simple Central Server Model (CSM). Thus a computation visits the CPU and an i/o device in succession one or more times. At all times a single PAWS transaction of category COMPT is allocated to a host. This transaction remains blocked at the ALLOCATE node HSTBLK as shown in Figure 4.9.

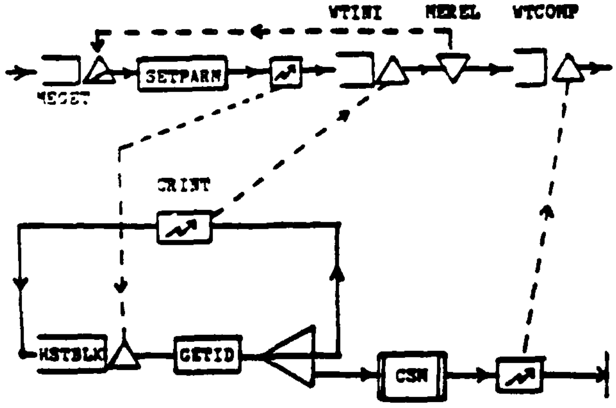


Figure 4.9. Computation on a Host

Figure 4.9 illustrates the instantiation and execution of the computation processes by some user of a host. Each host has a permanent transaction that is blocked indefinitely at HSTBLK. A host user first enters the mutual exclusion region bounded by the ALLOCATE node MEGET and the RELEASE node MEREL in the upper IPG. The user sets his identity and the type of computation he requires executed in some global variable. He then interrupts the transaction that represents the host he wishes to execute on at node HSTBLK. The interrupting transaction then blocks at WTINI. The interrupted host transaction gets the parameters stored in the variables and uses them to instantiate a sibling computation at GENCOMP. The host transaction then interrupts the host user transaction waiting at node WTINI at node CRINI. The host transaction then goes back to node HSTBLK to await the next interruption. The host user transaction exits the mutual exclusion region by releasing the token at MEREL and then goes to node WTCOMP to await termination of the initiated computation.

The initiated computation knows the identity of the host user transaction. On being created it performs the desired computation at the complex node CSM and then at node COMPFINI interrupts its corresponding host user transaction at node WTCOMP.

The IPG of Figure 4.9 illustrates how mutual exclusion can be modeled with PAWS. More importantly the host transaction is at any time the SIBLING of all of the computation transactions that are executing in the nodes of the CSM complex node. Thus it can interrupt all of them simultaneously without being aware of each of their identities. This is a fact that we will take advantage of in the following section.

4.5 A LOOK BACK

What have we accomplished so far? The chief benefit we have achieved is the ability to develop IPGs and thus simulation models of distributed systems in a clear top-down manner. In addition we have developed models for both hosts and local area networks that are intuitive and easy to understand. We next tackle the task of introducing failures into the model. ·

5. MODELING FAULTS IN DISTRIBUTED SYSTEMS

We next address the problem of modeling faults. There are a number of faults that can occur. Within a host, individual storage devices can fail. In addition, a host can fail due to CPU or main memory failure. In the network, if a link fails then some communication capacity will be lost. If a gateway fails, then all the links that are connected to it will fail. Finally, a cluster's local area network can fail, affecting all the communications that use the local area network to achieve communication between two clusters.

How do we introduce faults into the system? Faults can be modeled as a special transaction category. For each resource that can fail there will be an instance of this transaction category that generates and recovers failures according to a specified Time Between Failure (TBF) distribution and a Time To Repair (TTR) distribution. Obviously there will be different actions that need to be performed for each device failure. When a resource failure occurs all the transactions that were using the resource must be informed of the failure so that they can simulate the recovery actions in the system if necessary.

A fault transaction can be used to simulate a combination of related failures. For example, if the shared memory of a multiprocessor system fails then all the processors of the multiprocessor system will also fail. We now illustrate how we have modeled failures in the individual hosts.

5.1 COMPUTATION FAILURES

Recall from section 4.4 that a host consists of a set of resources and a set of related processes (modeled as PAWS transactions) executing on it. In Figure 4.9 we depicted this as the complex node CSM. In this section we expand that node into Figure 5.1.

We have assumed that our system is homogenous in that each host has the same configuration. We have done this in order to keep the simulation as simple as possible. Thus each host contains three discs represented as complex nodes.

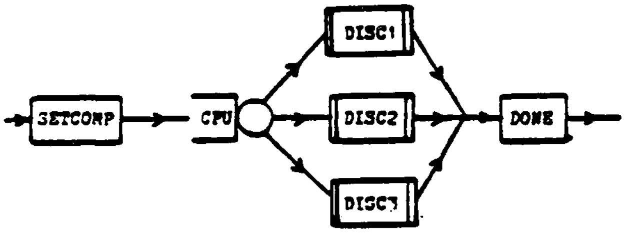


Figure 5.1. The Host Configuration

The transactions enter at node SETCOMP which decides the next phase of the computation. The phase of a computation is used to determine the requested CPU and disc service times and whether the computation has terminated. In Figure 5.2 we develop the complex node DISC1: the other secondary storage devices are similar.

Figure 5.2 consists of two interacting IPGs. One of these models is a fault transaction that causes the device to fail and recover. The other models the handling of computation transactions during both normal and failure modes.

The fault transaction waits for some TBF and then sets a global flag at node SETFLG, ensuring that all computation transactions that arrive at the device during failure will bypass it. Next the failure transaction uses the SETFLSH node to set the power of node DEV1 to be a very large number, ensuring that all the computation transactions waiting or receiving service at DEV1 will leave DEV1 immediately. Finally, the fault transaction will set DEV1's power to zero so as to block all subsequent requests. After a delay of some TTR the fault transaction will set the power of DEV1 back to 1 (normal operation) and reset the global flag.

Computation transactions go to nodes C1D1 to have their phase modified in case of failure. During normal operation the failure flag has been reset so the transaction will visit DEV1. During the failure of DEV1, the transactions will bypass the DEV1 node and leave the complex node DISC1 with a transaction phase that implies failure. Similarly C1D2 handles the flushed transactions that reach it immediately after a failure occurs.

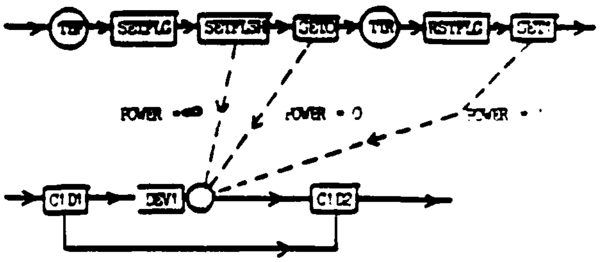


Figure 5.2. Model of the Storage Device

We finally address the notion of computation failures at the host level. In Figure 4.9 we described how each host possessed a single transaction. Further this transaction instantiated SIBLING computation transactions to execute computations on behalf of the host users. This approach pays off when modeling failures because the host's transaction can, when the host fails, be made to interrupt all of its siblings. These siblings may be anywhere in the CSM network at that instant of time. Their interruption implies that all computation at that host automatically ceases.

We illustrate this in Figure 5.3. Here the host user transaction IPG is not shown as in Figure 4.9. In Figure 5.3 the failure transaction for a host first waits for some interval before failure occurs. At failure the failure transaction enters the mutual exclusion region bounded by MEGET and MEREL. On entering this region it has mutually exclusive access to the host transaction of the host that must fail. It then sets its identity in global variables at node SETPARM transaction, interrupts the host transaction at HSTBLK and then waits at node WTINI.

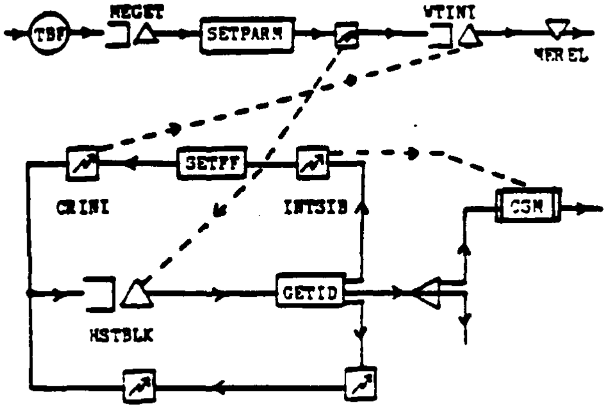


Figure 5.3. Host Failure Model

The host transaction is interrupted and gets the identity of its interruptor at GETID, and then at INTSIB interrupts all its siblings in CSM. Notice that this time it does not visit GENCOMP since it is not instantiating a new computation. After interrupting all its siblings, the host transaction sets the failure flag and then interrupts the failure transaction at node WTINI. The failure transaction exits the mutual exclusion region and then enters the TTR delay node. The recovery from failure simply requires that the failure flag be reset. If the host transaction is interrupted by a host user transaction while the failure flag is set, the host transaction interrupts the host user transaction and indicates that the computation has failed. This has been shown in Figure 5.3, which does not illustrate normal operation of the host transaction.

5.2. COMMUNICATIONS FAILURES

Failure of communications affects the static paths between clusters in the system. There is a hierarchy of failures in communications. A link failure implies that some set of paths between clusters fail. We call these paths the link's dependent set. Failure of a gateway host implies that the link dependent sets of all the links connected to that gateway fail. Finally a cluster failure implies that all the link dependent sets of all the links connected to all the gateways of that cluster have failed.

We define four data structures to hold the cluster information, the link information, the path information and the current connectivity information. All paths between any two clusters are ordered by increasing length. The current shortest path between two clusters is pointed to by the connectivity matrix.

Faults in the communiction system affect the long distance links or the local area networks. These cause packet transactions to be flushed out of these resources as in the models for device failure. Thus, the expanded IPGs for the LDLs and LANS will include nodes that force packet transactions to bypass the resource in case of failure. Figure 4.5 is extended to record transmission failure and the packet is re-transmitted after some delay by another route.

The PAWS USER node interface to FORTRAN was used to model communication failure in order to manipulate the data structures easily. The details of this model have currently been completed at IRA and will appear in a future report.

6. CONCLUSIONS

We have presented a practical modeling methodology for distributed systems encompassing models of fault propagation and fault recovery strategies. PAWS programs using these models have been implemented and executed. The major advantages of our methodology are 1) it is practical and usable by practitioners today, and 2) it is hierarchical, thus permitting easy modification of a complex model and permitting the modeling of a complex system to proceed in conjumction with the design of that system.

The RPC mechanism is, as we have mentioned earlier, a distributed programming primitive. This primitive is used by atomic database transactions in the ZEUS system design. The models for atomic actions will thus use the models we have developed in this paper.

We have also presented in this paper some important modeling techniques using PAWS. One of them is a clear intuitive model of a CSMA/CD network. Another is an effective means of integrating fault modeling with performance modeling.

ACKNOWLEDGEMENTS:

This work was partially funded by the Rome Air Development Center and was performed in conjunction with the Honeywell Corporation.

Discussions with Annette Palmer and Jim Dutton at IRA also contributed greatly to this project.

7. REFERENCES

[BIV84] Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls." ACM Trans. on Comp. Sys. 2.1 (February 1984), pp. 39-59.

[BRO84] Browne, J. C. et. al. "ZEUS: An Object-oriented High Integrity Distributed Operating System." (to appear in ACM Conference 1984).

[IRA84] Information Research Associates, Performance Analyst's Workbench System (PAWS) User's Manual, Austin, Texas 78705 (1984).

[MET76] Metcalfe, R. M. and Boggs, D. R. "Ethernet: Distributed Packet Switching for Local Computer Networks." Commun. ACM 19.7 (July 1976), pp. 237-255.

[SHR82] Shrivastava, S. K. and Panziera, F. "The Design of a Reliable Remote Procedure Call Mechanism." IEEE Trans. on Computers, Vol. C-31, 7 (July 1982), pp. 692-297.

[SPE82] Spector, A. Z. "Performing Remote Operations Efficiently on a Locl Computer Network." Commun. ACM 25.4 (April 1982), pp. 246-260.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.